Polylines

Often, we prototype our code in 2D before taking it to 3D, since it is usually faster and easier to debug. Therefore, it is very useful that our geometry processing toolbox contains enough 2D functionality to make quick prototyping easy. Furthermore, we want said functionality to be similar enough to the 3D one in the same toolboox that the leap from 2D to 3D can be done as seamlessly as possible. `gptoolbox` aims to satisfy these two needs.

When prototyping in 2D, however, it is useful to keep Alec Jacobson's words in mind, "what works in 3D must work in 2D, but what works in 2D does not necessarily work in 3D".

Get examples from the internet

Sometimes we just want to find a diverse set of shapes to try things on in 2D. Fortunately, `gptoolbox` allows us to get polyline curves from general png files. For example, here' an image of the beautiful Toronto skyline:

Tracing its polygonal outline in `gptoolbox` is as easy as running a one-liner:

```
>> [V,E] = png2poly('/data/toronto.png',0,Inf);
```

The output is in the form of a set of vertices `V` and a set of edges `E`, which is perfect for prototyping code we intend to ship later to 3D, since it matches the usual `V,F` format for triangle meshes. We can check that we have indeed obtained the right polyline by plotting the edges:

```
>> p = plot([V(E(:,1),1) V(E(:,2),1)]',[V(E(:,1),2) V(E(:,2),2)]');
```

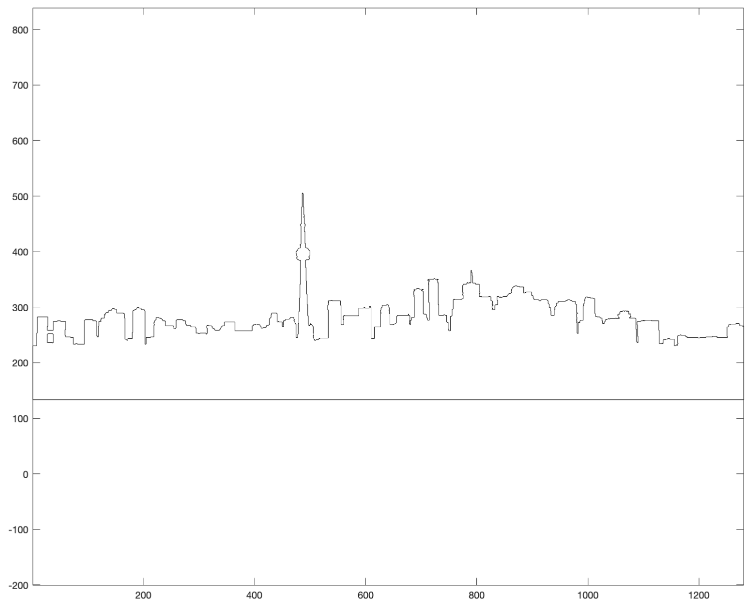
These vectorized expressions can be hard to remember and come up with on the spot, so `gptoolbox` also provides a wrapper that we can call in a similar way to `tsurf`:

```
>> plot_edges(V,E,'-k')
```

In both cases, we indeed obtain a polygonal representation of Toronto's skyline, which I can save, as usual, by running

```
>> figpng('toronto.png')
```

to get this:



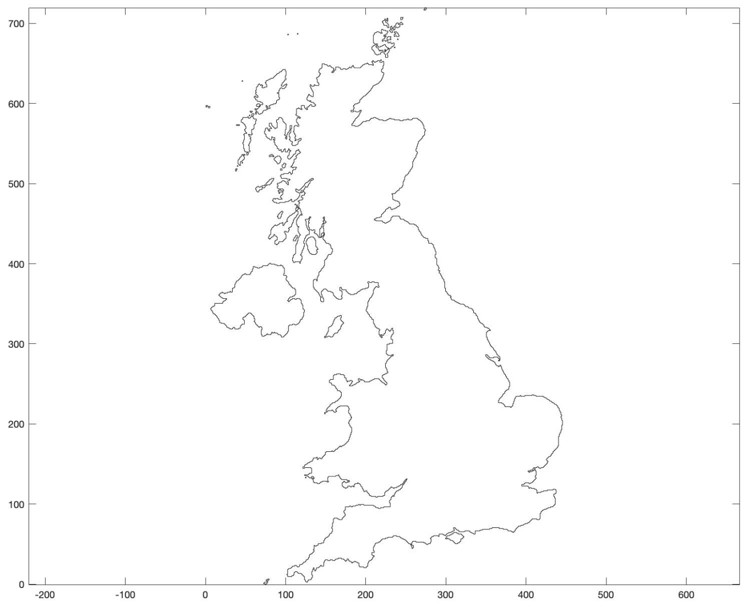
Note how there are no restrictions imposed on the topology of the output; `png2poly` traces all the boundary components of the png file.

You may also notice that our newly-acquired polyline has the same resolution that our original input image had. This is great, but it may not be desirable. For example, say I want a polyline of the silhouette of the United Kingdom, and I decide to use this image:

Then, calling

```
>> [V,E] = png2poly('/data/uk.png',0,Inf);
>> plot_edges(V,E,'-k')
```

will result in this:



It works like a charm! However, since this was a high-resolution input image, we can see there is a lot of detail in the polyline, which has over seven thousand vertices. If I am looking to prototype with this shape, I probably want to start with something simpler which has less of the noise and irregularities of this image. `gptoolbox` knows this, and that's why `png2poly` can accept a number of smoothing iterations as its second argument:

```
>> [V,E] = png2poly('/data/uk.png',10,Inf);
>> plot_edges(V,E,'-k')
```

which gives me a much simpler shape to test things on, with less irregularities:

700
600
500
400
300
200
100
-200
-100
0
100
200
300
400
500
600

This smoothing step is also useful when using very low resolution pngs, where the rasterization can include visible artifacts in the traced polyline. One or two smoothing iterations usually resolves this.

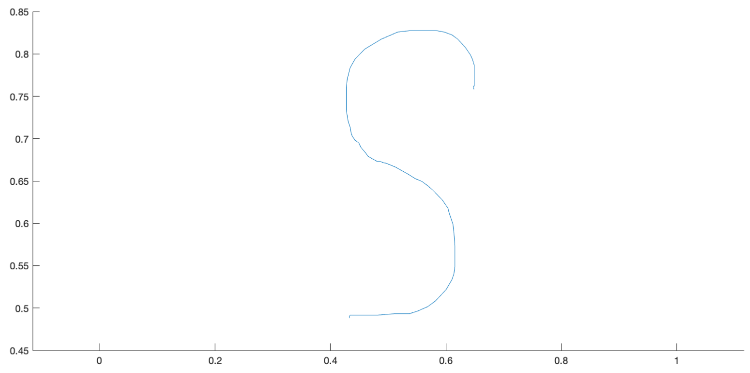
Draw your own examples

Getting examples from the internet can be a really fun way to create a diverse set of shapes in which to test your 2D code. However, the particular details of your project often means you or your collaborators have a very concrete idea of the key examples one would neeed to test a particular assumption or functionality. For these cases, `gptoolbox` allows you to directly draw a polyline inside a matlab figure.

Run

```
>> V = get_pencil_curve();
```

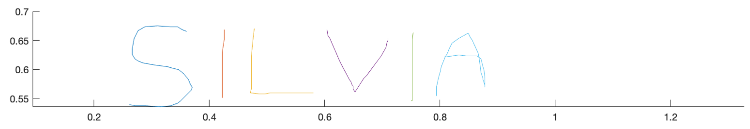
A figure window will appear (if it wasn't open already). Left-click and drag your cursor slowly on the figure window and you should start to see a curve appear, following your stroke. Stop holding the mouse button when you're happy with your curve.



As you may have noticed, `get_pencil_curve` only allows us to draw single component curves; meaning curves we can draw without lifting the pencil. Often, we want to create more complicated shapes. For that, `gptoolbox` provides us with `get_pencil_curves` (note the final `s`). Simply run

```
>> [V,E,cid] = get_pencil_curves(1e-6);
```

and, every type you end a stroke, type "y" into the matlab console to continue with another one. To end, type "n". That way, I can get something like this:



Exercises

Now it's time for you to get used to these functions. Why don't you use the skeleton scripts in `exercise/` to make polylines of shapes you like?