MATH 4500 Final Project

Fred Hohman
12-3-13

Goal: We want to minimize the distance of a path over the San Gabriel Mountains in California given a starting and ending point.

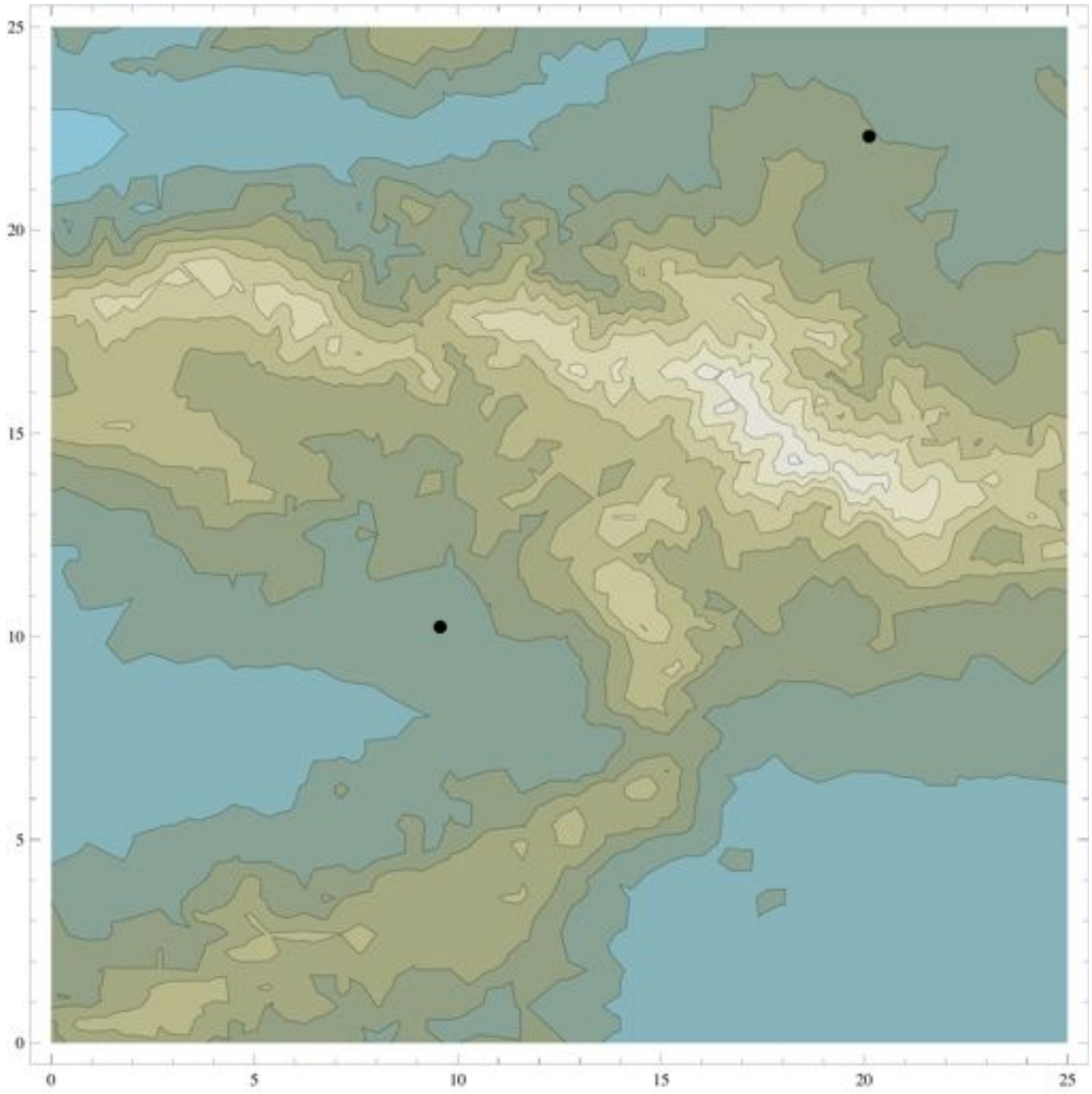
To begin, we will first need to import our external packages.

```
AppendTo[$Path, NotebookDirectory[]];
<< TerrainPackage`
```

This gives us the functions z[x,y] and GradZ[x,y].

To start things off, let's define our starting point and our ending point, and plot them on the contour map of z[x, y], just to get a feel of what we are after. Note: the map is in consistent units of kilometers.

```
StartPoint = {9.57, 10.23};
EndPoint = {20.12, 22.3};
ContourPlot[z[x, y], {x, 0, 25}, {y, 0, 25}, ColorFunction → "LightTerrain",
 ImageSize → Large, Epilog → {PointSize[Large], Point[{StartPoint, EndPoint}]}]
```



Task 1

Our first task is to have *Mathematica* take in a set of way points and return a pair of InterpolatingFunctions x and y so that a series of conditions is satisfied. To do this, I made the function WaypointInterpolatingFunction that takes in a set of points of the form {{x1, y1}, ... ,{xn, yx}} and gives back xoft and yoft. Since these outputted functions are parametrizations, I named my functions x and y to be xoft and yoft—just as a reminder of their t dependence, and to avoid confusion with the characters x and y.

```
WaypointInterpolatingFunction[points_] := {
  xoft =
   Module[{xlength},
    points[[All, 1]];
    xlength = Length[points[[All, 1]]];
    Range[0, 1, 1 / (xlength - 1)];
    Interpolation[
     Partition[Riffle[Range[0, 1, 1 / (xlength - 1)], points[[All, 1]]], 2]]
  ,
  yoft =
   Module[{ylength},
    points[[All, 2]];
    ylength = Length[points[[All, 2]]];
    Range[0, 1, 1 / (ylength - 1)];
    Interpolation[
     Partition[Riffle[Range[0, 1, 1 / (ylength - 1)], points[[All, 2]]], 2]]
 }
```

Now let's check that we have a set of InterpolatingFunctions, and that our given conditions are satisfied as defined in the instructions. To do this, simply take the WaypointInterpolatingFunction of our starting and ending point. The Quiet is there to hide the InterpolationOrder message. This is not an error, just Mathematica letting us know what it is doing behind the scenes.

```
Quiet[WaypointInterpolatingFunction[{StartPoint, EndPoint}]]
xoft[0]
yoft[0]
xoft[1]
yoft[1]
```

```
{InterpolatingFunction[{{0., 1.}}, <>], InterpolatingFunction[{{0., 1.}}, <>]}

9.57

10.23

20.12

22.3
```

We can now define a function p[t] to be used later.

```
p[t_] := {xoft[t], yoft[t]}
```

Task 2

We now want to define a function γ[t] that parametrizes the path in 3D space. This will also be used later.

```
ɣ[t_] := Join[p[t], {z[xoft[t], yoft[t]]}]
```

We now want to write our objective function called PathLength that should calculate the total length of the path in 3D space.

Throughout the project, I define and use a couple different versions of the PathLength function, all of which should be identified as PathLengthVersionName. This version of the PathLength below is for 3

way points, and “general”, i.e., I don’t use this since it won’t run because of the funny derivative of z[x[t], y[t]] situation. The version below is just see how other PathLength functions will be structured. The AccuracyGoal will be set low to ensure a fast evaluation time.

Also note that I define my γ functions outside of the PathLength function, which is a slight change from the instructions.

```
PathLength3PointsGeneral[x1_, y1_, x2_, y2_, x3_, y3_] :=
 NIntegrate[Norm[γ'[t]], {t, 0, 1}, AccuracyGoal → 1]
```

Task 3

We now want to run *Mathematica*’s NMinimize function on our PathLength function. However, to start, let’s run NMinimize where z[x,y]=0. This means we will only be minimizing over the x-y plane, so our shortest path should be the line between our starting and ending point.

But before we even do that, notice that we will have to type {{x1, y1}, {x2, y2}, ... {xn, yn}} a lot throughout these calculations. Instead, let’s have *Mathematica* generate it for us.

```
WaypointVariables[NumberOfPoints_] :=
  Table[ToExpression[# <> ToString[i]] & /@ {"x", "y"}, {i, 1, NumberOfPoints}];
```

So now let’s get a fresh set of InterpolatingFunctions xoft and yoft that are in terms of x1, y1, x2, yx2, x3, and y3.

```
WaypointInterpolatingFunction[Join[{StartPoint}, WaypointVariables[3], {EndPoint}]]
```

```
{InterpolatingFunction[{{0., 1.}}, <>], InterpolatingFunction[{{0., 1.}}, <>]}
```

Now we need to construct our γ function. Since we are only in the x-y plane, I made a new γ function called γ2D to represented the γ function that is only in 2 dimensions. Now recall our definition of p[t], and notice that p[t] = γ in the plane.

Once we have that, we can take the t derivative of γ2D with a *Mathematica* D function, since our z component is not present. This will produce another set of InterpolatingFunctions.

```
γ2D[t_] := p[t]
Dγ2D = D[γ2D[t], t]
```

```
{InterpolatingFunction[{{0., 1.}}, <>][t], InterpolatingFunction[{{0., 1.}}, <>][t]}
```

Now let’s make a new version of PathLength in the plane for only three points (like the instructions say).

```
PathLength2D[x1_, y1_, x2_, y2_, x3_, y3_] :=
 NIntegrate[Norm[Dγ2D], {t, 0, 1}, AccuracyGoal → 1]
```

It’s now time to run NMinimize on all 6 variables. I set limits on the variables so that the x values must stay in the domain {9,2 1} and the y values must stay in the range {10, 23}. I chose these by considering the coordinates of our starting and ending points. Also, we need to wrap our PathLength2D function in a Hold, so that Mathematica does not try to evaluate the integral before minimizing. I did not give the variables any initial values either, since it should converge to a line regardless.

I then call this solution TwoDPointsListWorking, to represent a list of points in 2D that I will perform

some operations on directly after running NMinimize.

```
TwoDPointsListWorking = NMinimize[Hold[PathLength2D[x1, y1, x2, y2, x3, y3]],
  {{x1, 9, 21}, {y1, 10, 23}, {x2, 9, 21}, {y2, 10, 23}, {x3, 9, 21}, {y3, 10, 23}},
  Method → {"SimulatedAnnealing"}, MaxIterations → 100, AccuracyGoal → 2]
```

```
{16.0308,
 {x1 → 10.3721, y1 → 11.1462, x2 → 12.6511, y2 → 13.7506, x3 → 17.4669, y3 → 19.2617}}
```

So we now have the path length of the straight line, and the values of which the variables converged too. We know that this is the correct path length, since the instructions state that we start 16.03km horizontally away from our ending point, which is approximately what we obtain.

Let's now take the second part of this list, and perform a transformation to extract the values of the solutions NMinimize gives. We can then Partition this list to obtain a set of couple points that minimizes the path length. Call this list TwoDPointsList, which represents our final list of points in 2D.

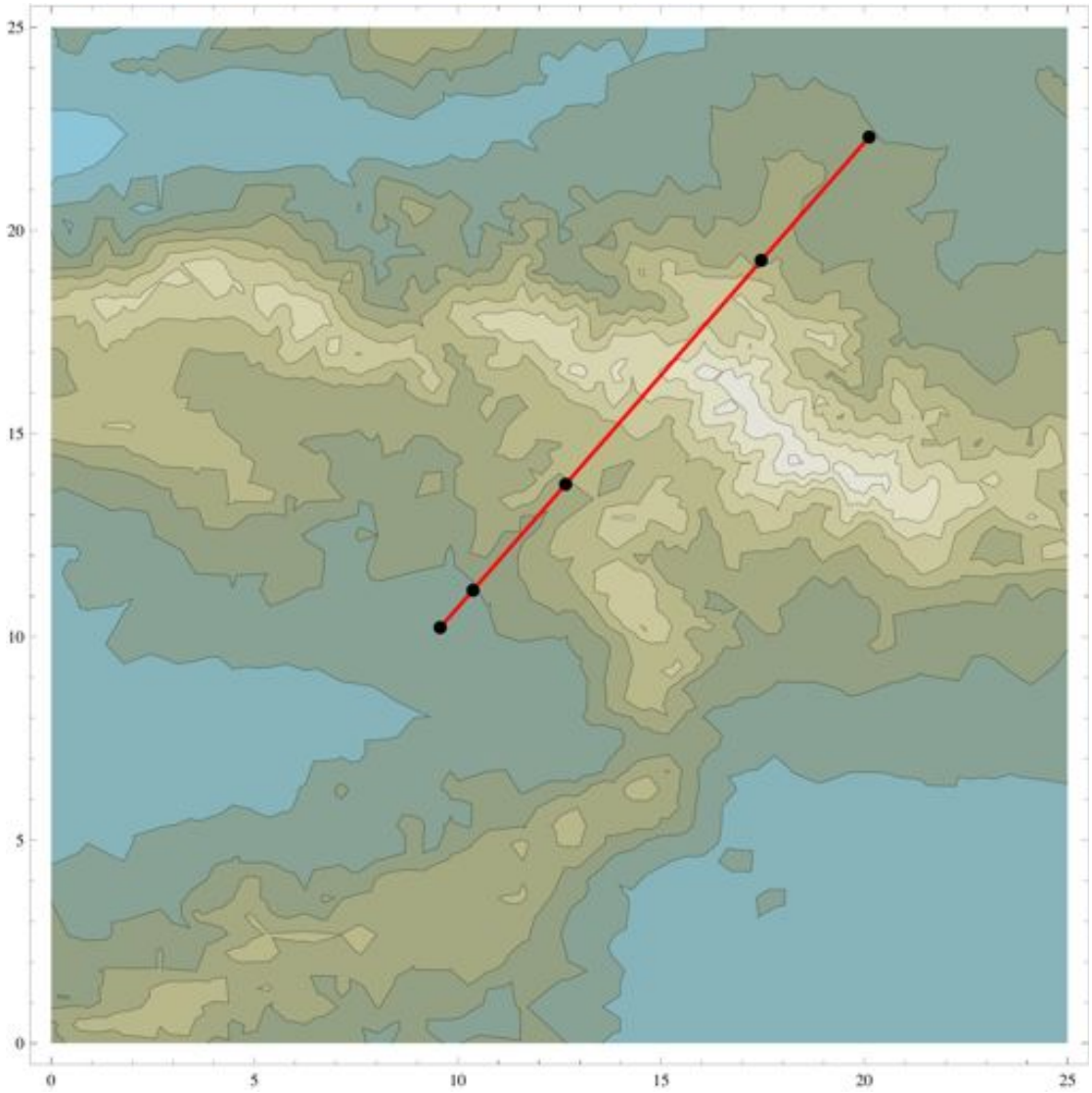
```
TwoDPointsList =
 Partition[Flatten[WaypointVariables[3]] /. TwoDPointsListWorking[[2]], 2]
```

```
{{10.3721, 11.1462}, {12.6511, 13.7506}, {17.4669, 19.2617}}
```

Next, we want a way to plot the path that our variable points define. To do this, I made the function PathPlot that takes in a set of points and plots the contour map of z from before, the starting and ending points, the points at which the variables converge to minimize the path length, and the path defined by the Interpolation of the converged variable points.

```
PathPlot[ListOfPoints_] := Show[
  ContourPlot[z[x, y], {x, 0, 25}, {y, 0, 25}, ColorFunction → "LightTerrain", Epilog →
    {PointSize[Large], Point[Join[{StartPoint}, ListOfPoints, {EndPoint}]]}],
  Plot[Interpolation[Join[{StartPoint}, ListOfPoints, {EndPoint}]][t],
   {t, 9.57, 20.12}, PlotStyle → {Directive[Red, Thick]}]
  , ImageSize → Large]
```

```
PathPlot[TwoDPointsList]
```



The contour map does not make much sense in this situation, since our path is only defined in the plane, but it is nice to see a visual of what is happening. Imagine that the red line cuts through the mountain and remains in one plane.

Task 4

Now it's time to run NMinimize on our 3D/original γ function. To test this, I tried a version where I minimized only 3 points (in order to keep my evaluation time lower) and a version with 10 points for my final answer.

3 Points.

I first got a fresh set of xoft and yoft Interpolating Functions. Now recall what our 3D γ functions is: γ[t] =

{x[t], y[t], z[x[t], y[t]]}. Since we need the derivative of γ with respect to t, I defined a new function Dγ to represent γ'[t]. The x and y components can be derived with a D function, but the z component needed to be the dot product of GradZ with {x'[t], y'[t]}. Next, I define a new version of PathLength function for 3 variable points in 3D space. Finally, I run NMinimize over my PathLength function for the 6 variables.

Notes about NMnimize:
• Used the same domain constraints where the x's are contained by {9, 21} and y's are contained from {10, 23}.
• Used Simulated Annealing with 100 MaxIterations and an AccuracyGoal of 2.
• Chose a set of initial points that were obtained from taking the integer components of the solutions to a previously run 3 point 3D NMinimize.
• Warning: This should take ~2 minutes to complete.

```
WaypointInterpolatingFunction[
  Join[{StartPoint}, WaypointVariables[3], {EndPoint}]];
γ[t_] := Join[p[t], {z[xoft[t], yoft[t]]}];
Dγ = {D[xoft[t], t],
   D[yoft[t], t],
   Dot[GradZ[xoft[t], yoft[t]], {D[xoft[t], t], D[yoft[t], t]}]};
PathLength3D[x1_, y1_, x2_, y2_, x3_, y3_] :=
  NIntegrate[Norm[Dγ], {t, 0, 1}, AccuracyGoal → 1, Method → "LocalAdaptive"];
```

```
ThreeDPointsListWorking3WP = NMinimize[Hold[PathLength3D[x1, y1, x2, y2, x3, y3]],
   {{x1, 9, 21}, {y1, 10, 23}, {x2, 9, 21}, {y2, 10, 23}, {x3, 9, 21}, {y3, 10, 23}},
   Method → {"SimulatedAnnealing", "InitialPoints" → {{11, 12, 14, 15, 16, 17}}},
   MaxIterations → 100, AccuracyGoal → 2] // Timing
```

```
{96.785958, {16.3822,
  {x1 → 11.1919, y1 → 11.9219, x2 → 13.9085, y2 → 14.8527, x3 → 15.9839, y3 → 17.3151}}}
```

So we now have the path length of the minimized path through 3 variable points, and the values of which the variables converged too. We want this number to be as close to 16.03 as possible.

Let's now take the second part of this list (twice, to ignore the Timing number), and perform a transformation to extract the values of the solutions NMinimize gives. We can then Partition this list to obtain a set of couple points that minimizes the path length. Call this list ThreeDPointsList3WP, which represents our final list of points in 3D for 3 way points.

```
ThreeDPointsListWorking3WP2 = ThreeDPointsListWorking3WP[[2]];
ThreeDPointsList3WP =
 Partition[Flatten[WaypointVariables[3]] /. ThreeDPointsListWorking3WP2[[2]], 2]
```

```
{{11.1919, 11.9219}, {13.9085, 14.8527}, {15.9839, 17.3151}}
```

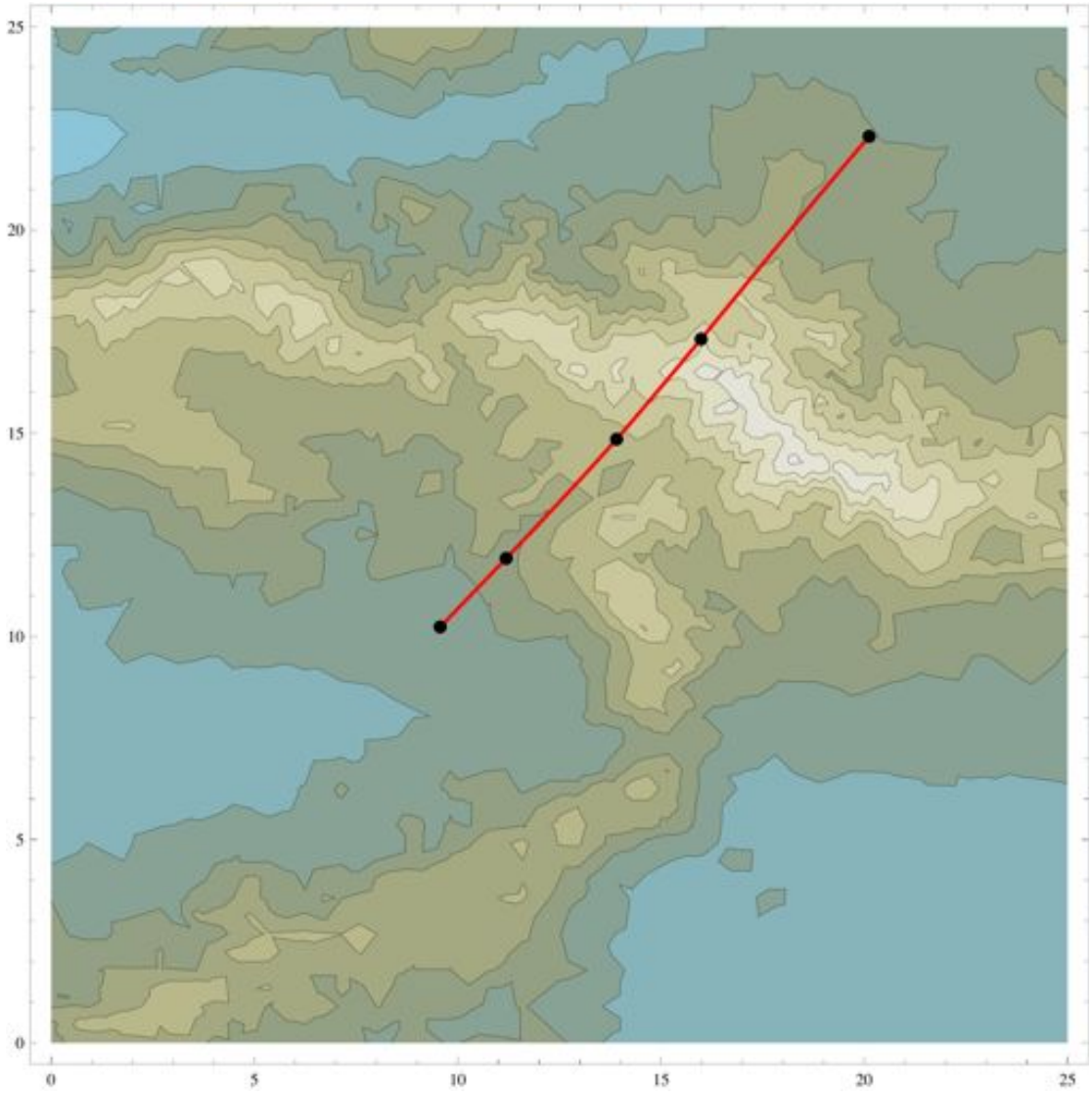
Let's now plot our path on the contour of z and also in 3D. Below is my PathPlot3D function, which plots the ParametricPlot3D of the Interpolation of the solution points we just obtained, and the 3D plot of z.

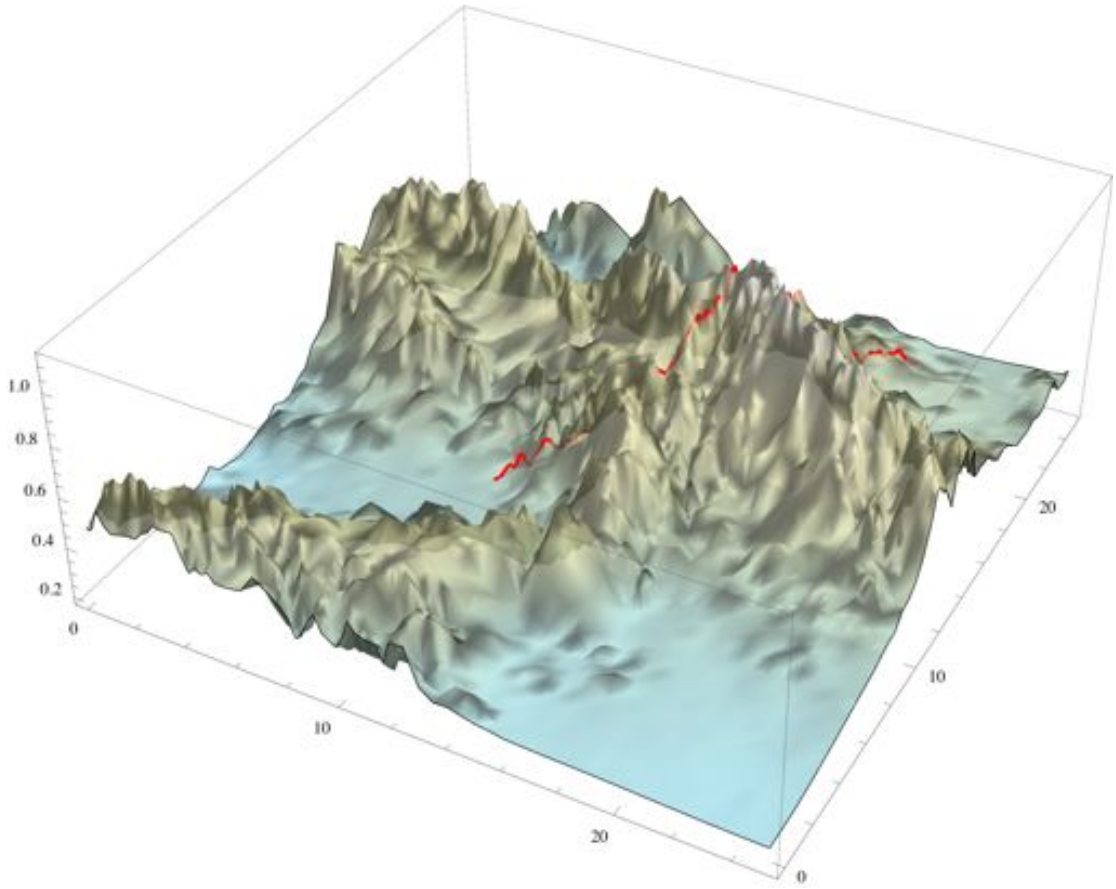
```
PathPlot3D[ListOfPoints_] := Show[
  Plot3D[z[x, y], {x, 0, 25}, {y, 0, 25}, ColorFunction → "LightTerrain",
   Epilog → {PointSize[Large], Point[Join[{StartPoint}, ListOfPoints, {EndPoint}]]},
   PlotPoints → 50, MaxRecursion → 2, Mesh → None, PlotStyle → {Opacity[0.8]}],
  ParametricPlot3D[γ[t], {t, 0, 1}, PlotStyle → {Directive[Red, Thick]}]
  , ImageSize → Large]
```

To use My PlotPath3D, we have to obtain a fresh set of xoft and yoft that are in terms of the solutions to

the NMinimize, i.e., no arbitrary variables such as x1, y1, etc. should be present. This is why the WaypointInterpolatingFunction is used again below.

```
PathPlot[ThreeDPointsList3WP]
WaypointInterpolatingFunction[
  Join[{StartPoint}, ThreeDPointsList3WP, {EndPoint}]];
PathPlot3D[ThreeDPointsList3WP]
```





10 Points.

Similarly, I first got a fresh set of xoft and yoft Interpolating Functions using 10 variable points. I then redefined Dγ in terms of the new γ function that is created when xoft and yoft are created. Next, I define a new version of PathLength function for 10 variable points in 3D space. Finally, I run NMinimize over my PathLength function for the 6 variables.

```
WaypointInterpolatingFunction[
  Join[{StartPoint}, WaypointVariables[10], {EndPoint}]];
Dγ = {D[xoft[t], t],
   D[yoft[t], t],
   Dot[GradZ[xoft[t], yoft[t]], {D[xoft[t], t], D[yoft[t], t]}]};
PathLength3D10WP[x1_, y1_, x2_, y2_, x3_, y3_, x4_, y4_,
  x5_, y5_, x6_, y6_, x7_, y7_, x8_, y8_, x9_, y9_, x10_, y10_] :=
 NIntegrate[Norm[Dγ], {t, 0, 1}, AccuracyGoal → 1, Method → "LocalAdaptive"]
```

The only drastic change from my 3 way point version is that I chose a specific set of initial points. These points will lie “directly” on the 2D line connecting the starting and ending point. So to generate this list, we first need the equation of the two dimensional line connecting our starting point and ending point.

```
Fit[{StartPoint, EndPoint}, {1, x}, x]
```

```
-0.718806 + 1.14408 x
```

So we can now make the function InitialPointsLine that is the equation of the line we want. So now let’s define our list of initial points in the correct format by Riffling a Range of x values with the same set of x

values passed through InitialPointsLine.

Aside: As I was copying and pasting the equation from the Fit function, I accidentally changed the y-intercept without knowing and ran my code with a different (but very similar) set of initial points. This accidental y-intercept actually cut my path length down by 0.02, so I am choosing to keep my "accidental" line equation. You can see the y-intercept is -0.776019 instead of -0.718806. Since these are just the initial guesses for our NMinimize, the points don't matter too much.

```
InitialPointsLine[x_] := 1.14692 x - 0.776019
InitialPointsList10WP =
 N[Riffle[Range[9, 21, (21 - 9) / 9], InitialPointsLine[Range[9, 21, (21 - 9) / 9]]]]
```

```
{9., 9.54626, 10.3333, 11.0755, 11.6667, 12.6047, 13., 14.1339, 14.3333, 15.6632,
 15.6667, 17.1924, 17., 18.7216, 18.3333, 20.2508, 19.6667, 21.7801, 21., 23.3093}
```

Finally, let's run NMinmize.

Notes about NMnimize:
• Used the same domain constraints where the x's are contained by {9, 21} and y's are contained from {10, 23}.
• Used Simulated Annealing with 100 MaxIterations and an AccuracyGoal of 2. NelderMead was faster, but not as accurate, and DifferentialEvolution was very slow.
• Chose a set of initial points that were obtained from picking equally spaced points on the line connecting the starting and ending point.
• Warning: This should take ~15-20 minutes to complete.

```
ThreeDPointsListWorking10WP =
 NMinimize[Hold[PathLength3D10WP[x1, y1, x2, y2, x3, y3, x4, y4, x5, y5, x6, y6,
     x7, y7, x8, y8, x9, y9, x10, y10]], {{x1, 9, 21}, {y1, 10, 23}, {x2, 9, 21},
    {y2, 10, 23}, {x3, 9, 21}, {y3, 10, 23}, {x4, 9, 21}, {y4, 10, 23}, {x5, 9, 21},
    {y5, 10, 23}, {x6, 9, 21}, {y6, 10, 23}, {x7, 9, 21}, {y7, 10, 23}, {x8, 9, 21},
    {y8, 10, 23}, {x9, 9, 21}, {y9, 10, 23}, {x10, 9, 21}, {y10, 10, 23}},
   Method → {"SimulatedAnnealing", "InitialPoints" → {InitialPointsList10WP}},
   MaxIterations → 100, AccuracyGoal → 2] // Timing
```

```
{949.432645,
 {16.373, {x1 → 9.7653, y1 → 10.4597, x2 → 10.0388, y2 → 10.7662, x3 → 11.8512,
   y3 → 12.8044, x4 → 13.05, y4 → 14.1844, x5 → 14.0476, y5 → 15.3984, x6 → 15.2684,
   y6 → 16.986, x7 → 17.3628, y7 → 19.045, x8 → 18.4664, y8 → 20.3701,
   x9 → 19.702, y9 → 21.8134, x10 → 19.9711, y10 → 22.1252}}}
```

So we now have the path length of the minimized path through 10 variable points, and the values of which the variables converged too. We also want this number to be as close to 16.03 as possible.

Similar as before, let's now perform a transformation to extract the values of the solutions NMinimize gives. Call this list ThreeDPointsList10WP, which represents our final list of points in 3D for 10 way points.

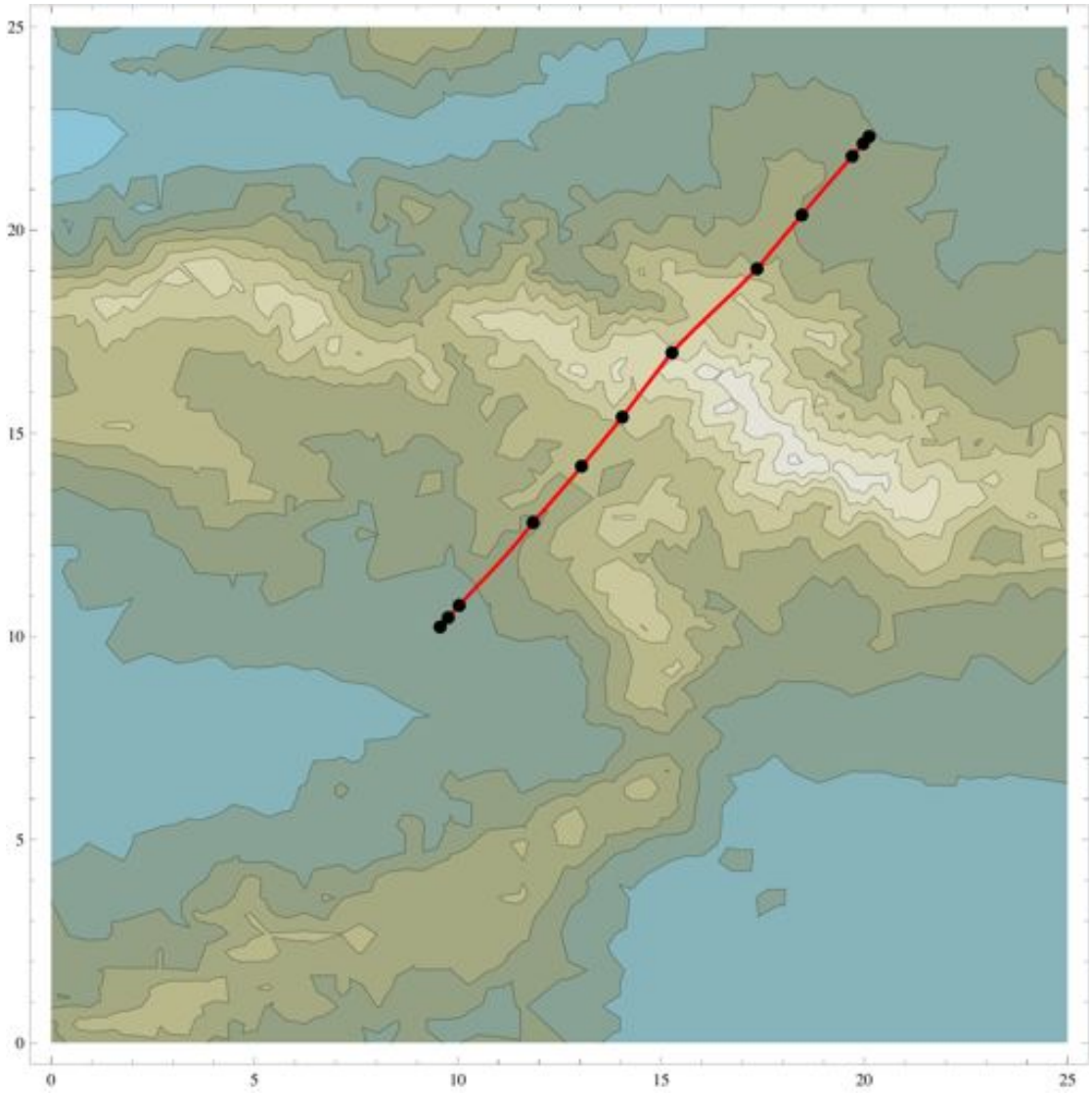
```
ThreeDPointsListWorking10WP2 = ThreeDPointsListWorking10WP[[2]];
ThreeDPointsList10WP = Partition[workinglist2 =
   Flatten[WaypointVariables[10]] /. ThreeDPointsListWorking10WP2[[2]], 2]
```

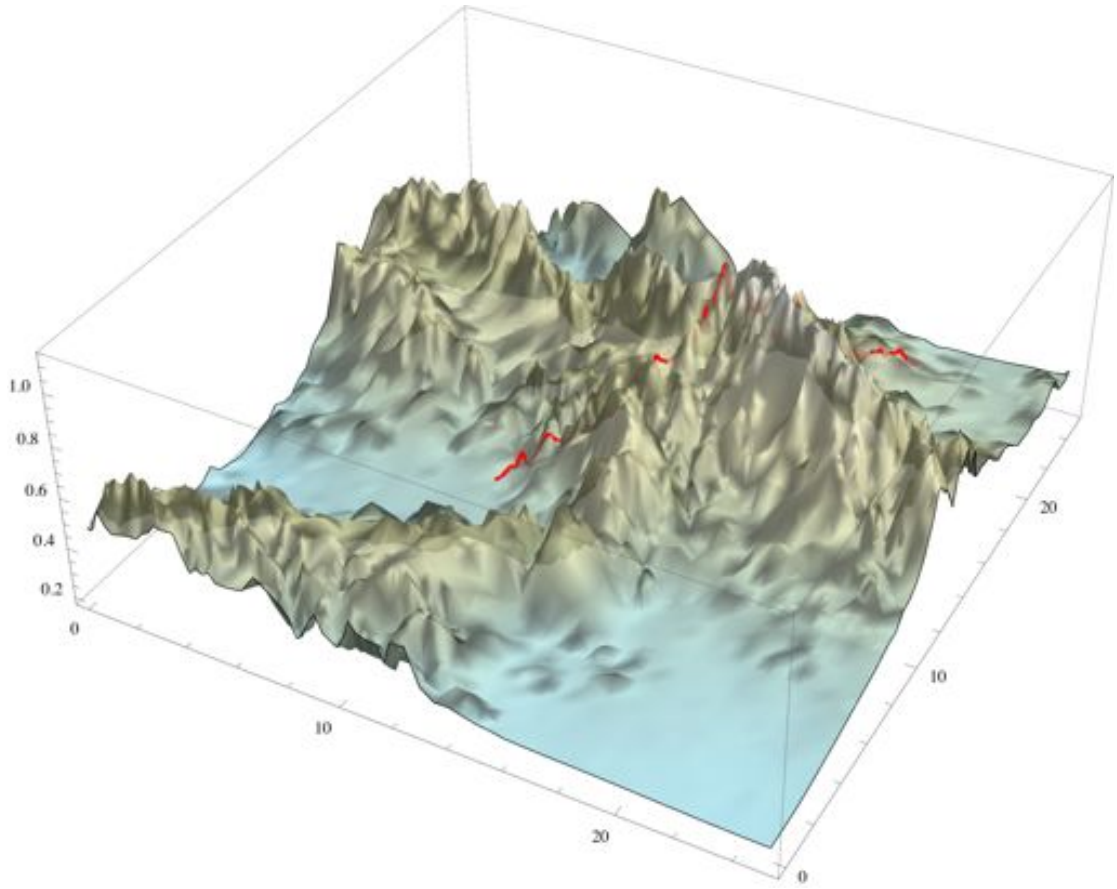
```
{{9.7653, 10.4597}, {10.0388, 10.7662}, {11.8512, 12.8044},
 {13.05, 14.1844}, {14.0476, 15.3984}, {15.2684, 16.986}, {17.3628, 19.045},
 {18.4664, 20.3701}, {19.702, 21.8134}, {19.9711, 22.1252}}
```

Let's now plot our path on the contour of z and also in 3D.

Recall that to use My PlotPath3D, we have to obtain a fresh set of xoft and yoft that are in terms of the solutions to the NMinimize, i.e., no arbitrary variables such as x1, y1, etc. should be present.

```
PathPlot[ThreeDPointsList10WP]
WaypointInterpolatingFunction[
  Join[{StartPoint}, ThreeDPointsList10WP, {EndPoint}]];
PathPlot3D[ThreeDPointsList10WP]
```





In conclusion, using 10 way points and my given NMinimize conditions I was able to find the shortest path length as approximately **16.373km**. The contour plot above shows our path in the plane, and the 3D plot shows our actual path over the mountain. Notice that our path in the 3D plot is not always on the surface z. This could lead to decimal path length error, as the path might go into the ground or jump above the ground at some points.