Block out points of roughness:

r = roughness bound that maximizes accuracy on training data
d = distance (rover width) in pixels that maximizes accuracy on training data
For **i** in **1**…**499**:
    For **j** in **1**…**499**:
        // Bound the points we check (in vertical and horizontal directions)
        top := row position a maximum distance **d** above **i**
        bot := row position a maximum distance **d** below **i**
        left := column position a maximum distance **d** left of **j**
        right := column position a maximum distance **d** right of **j**
        // Calculate number of rover positions that exist around point (i, j)
        ir := max(**1**, min(**i** - **top**, **bot** - **i**))
        jr := max(**1**, min(**j** - **left**, **right** - **j**))
        // Initialize multiplier arrays for calculating height at point x somewhere
        // between two other points
        im := [**1.**,…,**ir**] / (**ir** + **1**)
        jm := [**1.**,…,**jr**] / (**jr** + **1**)
        // Calculate expected height at point (i, j) for each rover position in the
        // vertical and horizontal ranges around (i, j)
        // Both * and + here are element-wise operations
        iheights := **im** * **DEM**[**top**:**top**+**ir**, **j**] + reverse(**im**) * **DEM**[**top**+**d**:**top**+**d**+**ir**]
        jheights := **jm** * **DEM**[**i**, **left**:**left**+**jr**] + reverse(**jm**) * **DEM**[**left**+**d**:**left**+**d**+**jr**]
        // If elevation at point (i, j) is more than r higher than any height in iheights
        // or jheights, block out the circle of points around (i, j)
        if **DEM**[**i**, **j**] - **r** > min(**iheights**) or **DEM**[**i**, **j**] - **r** > min(**jheights**):
            For **p** in getCircleIndices((**i***2, **j***2), **radius**, **resolution**):
                solution[**p**] = **0**

function getCircleIndices(**c**, **r**, **res**):
    points := new empty list
    r_new := math.ceil(**r**/**res**) // Convert radius in meters to radius in pixels
    For **i** in **c[0]**-**r_new**…**c[0]**+**r_new**:
        For **j** in **c[1]**-**r_new**…**c[1]**+**r_new**:
            // Optimization here: checks if solution at (i, j) is already 0 so we do
            // not spend time calculating euclidean distance unnecessarily
            if **solution**[**i**, **j**] == **255** and dist((**i**, **j**), **c**) <= **r**/**res**:
                points.append((**i**, **j**))

Optimizations:

We define the naive solution to be iterating over each point in the DEM, calling getCircleIndices with c = the current point, and checking if any of those points are taller than any of the planes formed by 3 of the border points. Assuming this checks a constant number of angles **A** to find the border points, getCircleIndices returns **C** points, and the DEM is size **NxN**, the naive solution would run in time O(ACN^2), which simplifies to O(N^2).

Our runtime in practice is much lower because we check if each point is the point of roughness, rather than if each possible circle contains a patch of roughness. We only call getCircleIndices if a point is rough, so the runtime is dependent on **X**, the number of points of that are raised/rough. We know X < N^2 and A is a large integer, so O(N^2 + CX) < O(ACN^2). However, because A and C are constants, our worse case runtime simplified is O(N^2 + X).

Block out points of extreme slope:

a = angle/slope bound that maximizes accuracy on training data
r = distance (rover width) in pixels that maximizes accuracy on training data
Parallelized outer loop using Pool.apply_async() from multiprocessing module:

```
For i in 10…490:
    For j in 10…490:
        // Skip calculations if height at point (i, j) is already at the minimum
        if DEM[i, j] == 0:
            continue
        points := list of 16 points spaced evenly in a circle of radius r around
            point (i, j)
        For p in 0…16:
            // Build plane using 3 points spaced 90 degrees apart
            p1 := points[p]
            p2 := points[(p + 4) % 16] // 90 degrees from p1
            p3 := points[(p + 8) % 16] // 180 degrees from p1
            vecA := vector from p2 to p1
            vecB := vector from p2 to p3
            n := crossproduct(vecA, vecB)
            angle := arccos(dotproduct(n, (0, 0, -1)) / norm(n))
            // If the angle between the plane's normal and the vertical > a,
            // block out the point
            // We don't need to check any more planes if this one is too sloped
            if angle > a:
                solution[2*i, 2*j] := 0
                solution[2*i + 1, 2*j] := 0
                solution[2*i, 2*j + 1] := 0
                solution[2*i + 1, 2*j + 1] := 0
                break
```

Optimizations:

The naive solution would assume we need to check every possible set of 3 landing points for the footpads of the rover, which would be at least the the circumference of the rover divided by average length in meters of a pixel in the DEM. Realistically, it would probably need to check more points to account for all possible positions where the tangent to the circle is not vertical or horizontal. This means checking at least 54 points. Our solution only checks 16 points, so we cut out at least

70% of the runtime that way.

We also chose to calculate the slopes using the plane normals, taking advantage of (and also optimizing for) the fact that the rover has 4 discreet footpads, rather than calculating the gradient of the plane or calculating the gradient at the central point. Calculating the gradient of any plane is slower than finding its normal, so avoiding that in general is preferable. Further, calculating gradient at the central point would be inaccurate due to the nature of the problem - the rover is landing on footpads around its edges, not a single leg in the middle.

Assuming the DEM is size **NxN**, the runtime of our solution is O(N^2).

Algorithm Outline:

1. Initialize (1000,1000) numpy array **solution** with zeros.
2. Set **solution**[20:979, 20:979] to 255 as points that may be allowable. Edges are not allowed.
3. Blocks out points of extreme slope.
4. Blocks out points of roughness.
5. Output solution to solutions/<datasetname>.pgm

False positive: Our algorithm marked a pixel as 255, should be 0
False negative: Our algorithm marked a pixel as 0, should be 255

Training data 1:

```
Time to find points of extreme slope: 9.80587601662
Searching for points of roughness...
Time to find points of roughness: 31.5083520412
False positives: 5016 = 2.5841816758 %
False negatives: 5756 = 0.71423608009 %
Total error: 10772 = 1.0772 %
```

not skipping dem = 0

```
False positives: 3934 = 2.02674854717 %
False negatives: 9213 = 1.14319961881 %
Total error: 13147 = 1.3147 %
```

Training data 2:

```
Time to find points of extreme slope: 35.0501070023
Searching for points of roughness...
Time to find points of roughness: 27.5057649612
False positives: 12066 = 1.54332594031 %
False negatives: 21732 = 9.96049169959 %
Total error: 33798 = 3.3798 %
```

without dem=0

```
False positives: 12066 = 1.54332594031 %
False negatives: 21732 = 9.96049169959 %
Total error: 33798 = 3.3798 %
```

Training data 3:

```
Time to find points of extreme slope: 24.6774759293
Searching for points of roughness...
Time to find points of roughness: 26.4193899632
False positives: 10286 = 1.17467292793 %
False negatives: 21310 = 17.1368373649 %
Total error: 31596 = 3.1596 %
```

without dem=0

```
False positives: 10286 = 1.17467292793 %
False negatives: 21310 = 17.1368373649 %
Total error: 31596 = 3.1596 %
```

Training data 4:

```
Time to find points of extreme slope: 21.1331930161
Searching for points of roughness...
Time to find points of roughness: 36.3402440548
False positives: 9149 = 1.00149419562 %
False negatives: 22395 = 25.9006534436 %
Total error: 31544 = 3.1544 %
```

without dem=0