1. (a) Let x_1 denote the amount of exterior paint produced and let x_2 denote the amount of interior paint produced. The problem is to:

$$\begin{aligned} \text{maximise} \quad & z = 5x_1 + 4x_2 \\ \text{subject to} \quad & 6x_1 + 4x_2 \leqslant 24 \\ & x_1 + 2x_2 \leqslant 6 \\ & x_2 - x_1 \leqslant 1 \\ & x_2 \leqslant 2 \\ & \mathbf{x} \geqslant \mathbf{0}. \end{aligned}$$

(b) Let r denote the radius of the paint can (cm) and let h denote the height of the paint can (cm). Note that the surface area of the cylindrical section is $2\pi rh + \pi r^2$, the surface area of the lid is πr^2, and the volume is $\pi r^2 h$. This results in a cost (in cents) of producing the paint can is $1.5(2\pi rh + \pi r^2) + 4\pi r^2 = 3\pi rh + 5.5\pi r^2$. The problem is to:

$$\begin{aligned} \text{minimise} \quad & z = 3\pi rh + 5.5\pi r^2 \\ \text{subject to} \quad & \pi r^2 h \geqslant 3500 \\ & r \leqslant 10 \\ & r, h \geqslant \mathbf{0}. \end{aligned}$$

(c) Let x_1 denote the amount of material sourced from Wood You Belive It, and let x_2 denote the amount of material sourced from Wood If I Could. The problem is to

$$\begin{aligned} \text{maximise} \quad & z = x_1 + x_2 \\ \text{subject to} \quad & 0.05x_1 + 0.01x_2 \leqslant 9 \\ & 50x_1 + 70x_2 \leqslant 15000 \\ & x_1 \leqslant 200 \\ & x_2 \leqslant 150 \\ & \mathbf{x} \geqslant \mathbf{0}. \end{aligned}$$

(d) Let x_1 denote the number of pens produced, and let x_2 denote the number of notepads produced. Note that every notepad goes into a pack. The problem is to

$$\begin{aligned} \text{maximise} \quad & z = 2x_1 + 3x_2 \\ \text{subject to} \quad & 2.5x_1 + x_2 \leqslant 18000 \\ & x_2 \geqslant 3000 \\ & 2x_1 - 3x_2 \geqslant 0 \\ & \mathbf{x} \geqslant \mathbf{0}. \end{aligned}$$

(e) Let $\mathbf{a} = (x, y)^T$ be a point on the parabola. The distance between $\mathbf{a}$ and the point $\mathbf{p}$ is

$$d(\mathbf{a}, \mathbf{p}) = \sqrt{x^2 + (1-y)^2}.$$

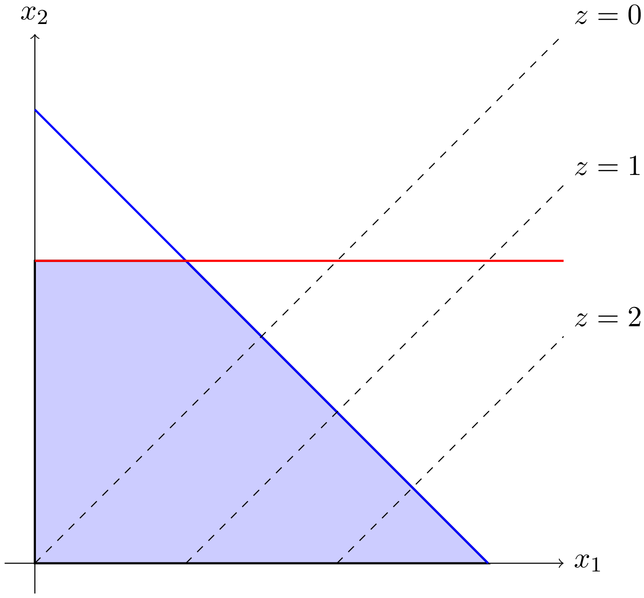
So the problem is to

$$\begin{aligned} \text{minimise} \quad & z = \sqrt{x^2 + (1-y)^2} \\ \text{subject to} \quad & y = x^2 - 1 \end{aligned}$$

Alternatively, by substituting $y = x^2 - 1$ into $d(\mathbf{a}, \mathbf{p})$, we get $z = \sqrt{x^2 + (1-(x^2-1))^2} = \sqrt{x^4 - 3x^2 + 4}$, which makes the problem unconstrained:

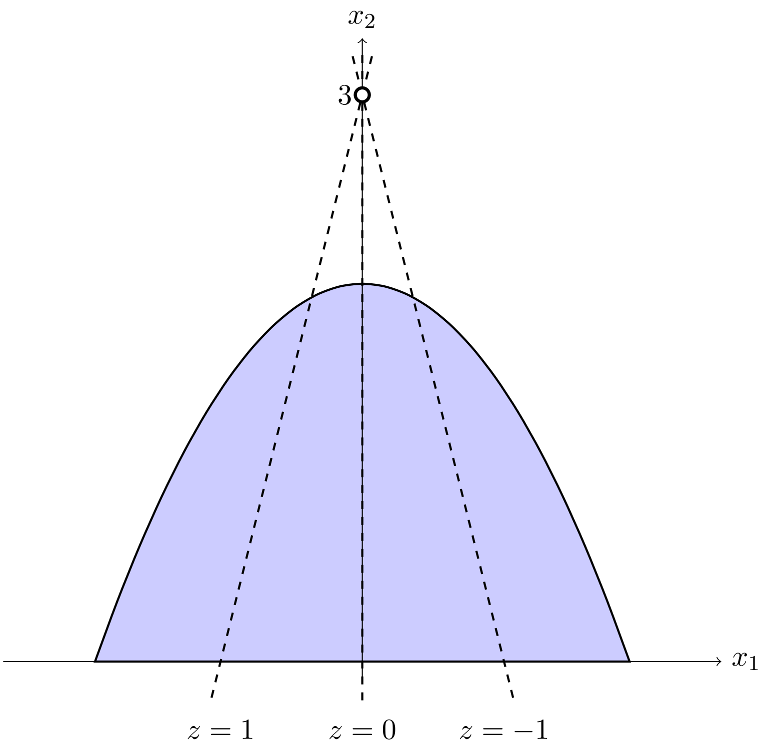
$$\text{minimise} \quad z = \sqrt{x^4 - 3x^2 + 4}$$

2. The feasible region and level sets are shown below.



As the level sets of z increase, they sweep across the feasible region and "leave" at the corner point $(x_1, x_2) = (3, 0)$. The maximum of z is attained at this corner point, giving $z = 3$ when $\mathbf{x}^T = (3, 0)$.

3. The feasible region and level sets are shown below.



Notice that as z increases, the level sets are "rotating" clockwise about the point $(0, 3)$. This suggests that the optimal value occurs at a point on the parabola where its tangent line passes through the point $(0, 3)$. One can show that this occurs at the point $(x_1, x_2) = (1, 1)$ giving $z = 2$.

4. There are four local minimisers of f, and they look like they are near the coordinates $(-4, -3)$, $(-3, 3)$, $(4, -2)$ and $(3, 2)$. This is not a very good way to solve these problems.

5. The surface plot should make the local minimisers stand out a little more, and it still appears that there are four local minimisers.

6. Running `fminsearch(f, [0,0])` finds a local minimiser at $(x_1, x_2) = (3, 2)$.

   To find the other local minimisers, we should start the search close to those speculated in Question 4.

   - Running `fminsearch(f, [-4,-3])` finds a local minimum at approximately $(x_1, x_2) = (-3.7793, -3.2832)$.
   - Running `fminsearch(f, [-4,3])` finds a local minimum at approximately $(x_1, x_2) = (-2.8051, 3.1313)$.
   - Running `fminsearch(f, [4,-2])` finds a local minimum at approximately $(x_1, x_2) = (3.5844, -1.8481)$.

7. (a) `X = randi([-5,5], 1, randi([10,20], 1, 1));`

   (b) `sum(X)`

   (c) `X(1) * X(end)`

   (d) `sum(X(mod(X,3) == 0))`

   (e) `X(X ~= max(X))`

   (f) `X(X >= mean(X))`

   (g)
   ```
   size = 2*length(X);
   duplicates = zeros(1, size);
   duplicates(1:2:size) = X;
   duplicates(2:2:size) = X;
   ```

8. A “direct” approach:

```
function [roots, discriminant] = solveQuadratic(a,b,c)
discriminant = b^2 - 4*a*c;
roots = [(-b-sqrt(discriminant))/(2*a), (-b+sqrt(discriminant))/(2*a)];
if discriminant < 0
    % Complex roots; ensure sorted correctly.
    r1 = roots(1);
    r2 = roots(2);
    if (imag(r1) > imag(r2))
        roots = [r2 r1];
    end
elseif discriminant > 0
    % Real roots; ensure sorted correctly.
    roots = sort(roots);
end
end
```

A more subtle approach:

```
function [roots, discriminant] = solveQuadratic(a, b, c)
% Multiplying the equation ax^2+bx+c = 0 by -1 will give the same solutions.
% If a > 0, then arranging roots with -sqrt(...) first and +sqrt(...) second
% will ensure the conditions are met.
if a < 0
    a = a*-1;
    b = b*-1;
    c = c*-1;
end
discriminant = b^2 - 4*a*c;
roots = [(-b-sqrt(discriminant))/(2*a), (-b+sqrt(discriminant))/(2*a)];
end
```

9. All that is necesary is to adjust the arguments of the function, and add lines defining `a`, `b` and `c` in terms of the new `coefficients` argument. For example:

```
function [roots, discriminant] = solveQuadratic(coefficients)
a = coefficients(1);
b = coefficients(2);
c = coefficients(3);
if a < 0
    a = a*-1;
    b = b*-1;
    c = c*-1;
end
discriminant = b^2 - 4*a*c;
roots = [(-b-sqrt(discriminant))/(2*a), (-b+sqrt(discriminant))/(2*a)];
end
```

10. (a)
```
size = randi([5,10],1,1);
A1 = randi([-5,5], 1, size);
A2 = randi([-5,5], 1, size);
```

(b) `A = [A1 A2];`

(c)
```
C = zeros(1, 2*size);
C(1:2:2*size) = A1;
C(2:2:2*size) = A2;
```

(d)
```
doubles = [];
for i = A
    doubles = [doubles sum(A == i) == 2];
end
```

(e)
```
uniques = [];
for i = A
    if ~ismember(i, uniques)
        uniques = [uniques i]
    end
end
```

(f)
```
counts = [];
for i = A
    counts = [counts sum(A == i)];
end
```

(g)
```
doubles = A(counts == 2);
uniques = A(counts == 1);
```

(h) Comparing `A` with its transpose `A'` will construct a square matrix where the entry in row `i`, column `j` is the result of comparing `A(i)` with `A(j)`. Summing together row `i` of this matrix will count the number of times `A(i)` appears in `A`. So the following code will do the job:

```
counts = sum(A == A');
```

This approach tends to be faster than using a loop, but this is at the cost of using more space in memory.