Research Skills Problem Set 2, Michelle Gurevich

Question 1

(a) Look up the help pages for the Mathematica functions `Grad`, `Div`, `Curl` and `Laplacian`, and hence calculate

$\nabla\phi$ and $\nabla^2\phi$, where $\phi(x, y) = x^3 - 3xy^2$;

$\nabla.\vec{v}$ and $\nabla\times\vec{v}$, where $\vec{v}(x, y) = (x^2 - y^2, 2xy)^T$

```
In[2]:= ϕ[x_, y_] := x^3 - 3 x y^2
```

$\nabla\phi$ is calculated

```
In[3]:= Grad[ϕ[x, y], {x, y}]
```

$\nabla^2\phi$ is calculated

```
In[4]:= Laplacian[ϕ[x, y], {x, y}]
```

```
In[5]:= v = {x^2 - y^2, 2 x y, 0}
```

$\nabla.\vec{v}$ is calculated

```
In[6]:= Div[v, {x, y, z}]
```

$\nabla\times\vec{v}$ is calculated

```
In[7]:= Curl[v, {x, y, z}]
```

(b) The inactive forms of these functions can be used to solve partial differential equations. For example, the following solves Laplace's equation on the region $0 \le x \le 1$, $0 \le y \le 1$ with boundary conditions $u(x, 0) = u(0, y) = u(x, 1) = 0$, $u(1, y) = \sin(\pi y)$:

```
In[8]:= eqn = Inactive[Laplacian][u[x, y], {x, y}] == 0
```

```
In[9]:= sol = NDSolve[
          {eqn, u[x, 0] == u[0, y] == u[x, 1] == 0, u[1, y] == Sin[Pi y]}, u, {x, 0, 1}, {y, 0, 1}]
```

Create a surface plot, a density plot and a contour plot of this solution.

```
In[10]:= Plot3D[u[x, y] /. sol, {x, 0, 1}, {y, 0, 1},
          AxesLabel → Automatic, PlotLabel → "Surface Plot"]
```

```
In[11]:= DensityPlot[u[x, y] /. sol, {x, 0, 1}, {y, 0, 1},
          FrameLabel → Automatic, PlotLabel → "Density Plot"]
```

```
In[12]:= ContourPlot[u[x, y] /. sol, {x, 0, 1}, {y, 0, 1},
          FrameLabel → Automatic, PlotLabel → "Contour Plot"]
```

(c) Solve the same PDE, with the same conditions, using `DSolve` instead, and compare your solutions graphically.

There is an edge effect in these graphs possibly due to a resolution issue, but except for these regions the graphs generated using the inactive forms and DSolve agree with each other.

```
In[13]:= sol2 = DSolve[
           {eqn, u[x, 0] == u[0, y] == u[x, 1] == 0, u[1, y] == Sin[Pi y]}, u, {x, 0, 1}, {y, 0, 1}]

In[14]:= Plot3D[u[x, y] /. sol2, {x, 0, 1}, {y, 0, 1},
          AxesLabel → Automatic, PlotLabel → "Surface Plot"]

In[15]:= DensityPlot[u[x, y] /. sol2, {x, 0, 1}, {y, 0, 1},
          FrameLabel → Automatic, PlotLabel → "Density Plot"]

In[16]:= ContourPlot[u[x, y] /. sol2, {x, 0, 1}, {y, 0, 1},
          FrameLabel → Automatic, PlotLabel → "Contour Plot"]
```

(d) The boundary conditions $u(x, 0) = u(0, y) = u(x, 1) = 0$, $u(1, y) = \sin(\pi y)$ can be replaced by the single **Dirichlet condition** $u = x \sin(\pi y)$, expressed in Mathematica as `DirichletCondition[u[x,y] == x Sin[π y], True]`. Try this in the `NDSolve` variation, confirming that it works. (It doesn't really work with `DSolve`).

```
In[17]:= solD = NDSolve[
           {eqn, DirichletCondition[u[x, y] == x Sin[Pi y], True]}, u, {x, 0, 1}, {y, 0, 1}]
```

(e) Solve the same equation, with the same Dirichlet condition, on the unit disk instead of the unit square, by replacing `{x,0,1},{y,0,1}` with **Element[{x, y}, Disk[]]**

```
In[18]:= sol3 = NDSolve[
           {eqn, DirichletCondition[u[x, y] == x Sin[Pi y], True]}, u, Element[{x, y}, Disk[]]]
```

In[•]:= Show your result graphically.

```
In[19]:= Plot3D[Evaluate[u[x, y] /. sol3], Element[{x, y}, Disk[]],
          AxesLabel → Automatic, PlotLabel → "Surface Plot"]

In[20]:= DensityPlot[u[x, y] /. sol3, Element[{x, y}, Disk[]],
          FrameLabel → Automatic, PlotLabel → "Density Plot"]

In[21]:= ContourPlot[u[x, y] /. sol3, Element[{x, y}, Disk[]],
          FrameLabel → Automatic, PlotLabel → "Contour Plot"]
```

(f) Solve numerically the simplified convection-diffusion equation for incompressible convection flows, $\frac{\partial u}{\partial t} == D \nabla^2 u - \vec{\nabla} . \nabla u$ on the domain $-1 \leq x \leq 1, -1 \leq y \leq 1$, for $0 \leq t \leq 1$, subject to the initial condition $u(0, x, y) = \sin(\pi x y)$ and the Dirichlet boundary condition $u(t, x, y) = \sin(\pi x y)$. Here, D should have the value 0.1, and the convection field $\vec{\nabla}$ should be `{y, -x}`.

```
In[22]:= ClearAll;
         d = .1;
         v = {y, -x};

In[25]:= diffusionEqn := Inactivate[D][u[x, y, t], t] -
            d Inactive[Laplacian][u[x, y, t], {x, y}] + v.Inactive[Grad][u[x, y, t], {x, y}] == 0

In[26]:= diffusionSoln = NDSolve[{diffusionEqn, u[x, y, 0] == Sin[Pi x y],
            DirichletCondition[u[x, y, t] == Sin[Pi x y], True]},
           u, {x, -1, 1}, {y, -1, 1}, {t, 0, 1}]
```

(g) Create a diagram showing a contour plot of your solution at $t = 1$, on which you should superimpose a vector plot of the convection field.

```
In[27]:= CP1 := ContourPlot[Evaluate[u[x, y, 1] /. diffusionSoln], {x, -1, 1}, {y, -1, 1} ,
           FrameLabel → Automatic, PlotLabel → "Diffusion solution with t=1"]
         VP1 := VectorPlot[v, {x, -1, 1}, {y, -1, 1}]
         PartG = Show[CP1, VP1]
```

(h) Repeat (f) and (g), this time on the unit disk.

```
In[30]:= diffusionSolnDisk =
          NDSolve[{diffusionEqn, u[x, y, 0] == Sin[Pi x y], DirichletCondition[
             u[x, y, t] == Sin[Pi x y], True]}, u, {t, 0, 1}, Element[{x, y}, Disk[]]]

In[31]:= CP1 := ContourPlot[Evaluate[u[x, y, 1] /. diffusionSoln], Element[{x, y}, Disk[]],
           FrameLabel → Automatic, PlotLabel → "Diffusion solution on unit disk"]
         VP1 := VectorPlot[v, Element[{x, y}, Disk[]]]
         PartH = Show[CP1, VP1]
```

(i) Export your diagrams from parts (g) and (h) as PNG files, and also as PDF files; when you submit this assignment, bundle them in a ZIP file together with your answer notebook.

```
In[34]:= Export["PartG.png", PartG]
         Export["PartH.png", PartH]
         Export["PartG.pdf", PartG]
         Export["PartH.pdf", PartH]
```

Question 2

Given a rectangular array of numbers ,

$$\begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & a_{2,3} & \cdots & a_{2,n} \\ a_{3,1} & a_{3,2} & a_{3,3} & \cdots & a_{3,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & a_{m,3} & \cdots & a_{m,n} \end{pmatrix},$$

the **von Neumann neighbourhood** of $a_{i,j}$ consists of the entries $\{a_{i-1,j}, a_{i,j-1}, a_{i+1,j}, a_{i,j+1}\}$.

There is a decision to be made about what to do at the boundaries of the array; for this question, we'll *wrap around*, so that for example the von Neumann neighbourhood of $a_{1,3}$ is $\{a_{m,3},\ a_{1,2},\ a_{2,3}, a_{1,4}\}$, and that of $a_{1,1}$ is $\{a_{m,1}, a_{1,n}, a_{2,1},\ a_{1,2}\}$.

(a) Write and test a function called `vonNeumannNeighbourhood`, which takes as its arguments a rectangular array `mat`, and two indexes `i` and `j`, and returns the von Neumann neighbourhood of `mat[[i, j]]`.

```
In[38]:= mat = Table[Subscript[a, i, j], {i, 6}, {j, 7}];
         mat // MatrixForm
         (* vonNeumannNeighbourhood[mat, i_, j_] :=
          {mat[[i-1,j]], mat[[i,j-1]], mat[[i+1,j]], mat[[i,j+1]]}
           vonNeumannNeighbourhood[mat, 4,2] *)
```

```
In[43]:= Clear[vonNeumannNeighbourhood];
         vonNeumannNeighbourhood[mat_, i_, j_] := Module[{
             rows = Dimensions[mat][[1]],
             columns = Dimensions[mat][[2]]},
           Which[
             (* inner case *)
             i != 1 && i != rows && j != 1 && j != columns,
             output = {mat[[i - 1, j]], mat[[i, j - 1]], mat[[i + 1, j]], mat[[i, j + 1]]},
             (* top left corner *)
             i == 1 && j == 1,
             output = {mat[[rows, j]], mat[[i, columns]], mat[[i + 1, j]], mat[[i, j + 1]]},
             (* top right corner *)
             i == 1 && j == columns,
             output = {mat[[rows, j]], mat[[i, j - 1]], mat[[i + 1, j]], mat[[i, 1]]},
             (* bottom left corner *)
             i == rows && j == 1,
             output = {mat[[i - 1, j]], mat[[i, columns]], mat[[1, j]], mat[[i, j + 1]]},
             (* bottom right corner *)
             i == rows && j == columns,
             output = {mat[[i - 1, j]], mat[[i, j - 1]], mat[[1, j]], mat[[i, 1]]},
             (* top neighbor wrap around *)
             i == 1 && j != 1 && j != columns,
             output = {mat[[rows, j]], mat[[i, j - 1]], mat[[i + 1, j]], mat[[i, j + 1]]},
             (* left neighbor wrap around *)
             i != 1 && i != rows && j == 1,
             output = {mat[[i - 1, j]], mat[[i, columns]], mat[[i + 1, j]], mat[[i, j + 1]]},
             (* right neighbor wrap around *)
             i != 1 && i != rows && j == columns,
             output = {mat[[i - 1, j]], mat[[i, j - 1]], mat[[i + 1, j]], mat[[i, 1]]},
             (* bottom neighbor wrap around *)
             i == rows && j != columns && j != 1,
             output = {mat[[i - 1, j]], mat[[i, j - 1]], mat[[1, j]], mat[[i, j + 1]]}];
           output]

         vonNeumannNeighbourhood[{{1, 3, 5}, {2, 4, 6}, {3, 5, 7}}, 2, 2]

In[46]:= vonNeumannNeighbourhood[{{1, 3, 5}, {2, 4, 6}, {3, 5, 7}}, 1, 1]

In[47]:= {{1, 3, 5}, {2, 4, 6}, {3, 5, 7}} // MatrixForm

In[48]:= vonNeumannNeighbourhood[{{1, 3, 5}, {2, 4, 6}, {3, 5, 7}}, 1, 3]

In[49]:= vonNeumannNeighbourhood[{{1, 3, 5}, {2, 4, 6}, {3, 5, 7}}, 2, 3]
```

(b) A forest fire is modelled as follows. Each space in the forest corresponds to a point in a rectangular lattice. The point may be in state 0 (“empty”), 1 (“tree”) or 2 (“burning tree”).

The following, for example, represents a sparsely wooded 10 × 10 forest, with one burning tree. Empty space is shown as purple, trees are shown as green, and burning trees are shown as red.

```
In[50]:= mat = {{1, 1, 0, 0, 1, 0, 0, 1, 0, 0}, {0, 0, 1, 0, 1, 1, 1, 0, 1, 0},
            {1, 0, 1, 1, 1, 1, 1, 1, 0, 0}, {0, 0, 0, 1, 0, 0, 0, 1, 0, 1},
            {0, 0, 0, 1, 2, 0, 0, 0, 1, 0}, {1, 0, 0, 0, 1, 0, 0, 0, 0, 1},
            {0, 0, 1, 0, 0, 0, 1, 1, 1, 1}, {1, 0, 0, 1, 1, 0, 0, 0, 1, 0},
            {1, 0, 0, 0, 0, 0, 0, 1, 0, 0}, {1, 1, 0, 0, 0, 0, 1, 1, 1, 1}};

In[51]:= MatrixPlot[mat / 2, PlotRange → {0, 1},
          ColorFunction -> "Rainbow", ColorFunctionScaling → False]
```

At each time step:

any point in state “empty” remains “empty”;

any point in state “burning tree” becomes “empty”;

any point in state “tree” remains at “tree” if there are no burning trees in its von Neumann neighbourhood, but if there are any, it becomes “burning tree”.

Define, and illustrate using `MatrixPlot`, a rectangular array `mat1` representing the forest after one time step.

```
In[52]:= mat1 = {{1, 1, 0, 0, 1, 0, 0, 1, 0, 0}, {0, 0, 1, 0, 1, 1, 1, 0, 1, 0},
            {1, 0, 1, 1, 1, 1, 1, 1, 0, 0}, {0, 0, 0, 1, 0, 0, 0, 1, 0, 1},
            {0, 0, 0, 2, 0, 0, 0, 0, 1, 0}, {1, 0, 0, 0, 2, 0, 0, 0, 0, 1},
            {0, 0, 1, 0, 0, 0, 1, 1, 1, 1}, {1, 0, 0, 1, 1, 0, 0, 0, 1, 0},
            {1, 0, 0, 0, 0, 0, 0, 1, 0, 0}, {1, 1, 0, 0, 0, 0, 1, 1, 1, 1}};
         MatrixPlot[mat1 / 2, PlotRange → {0, 1}, ColorFunction -> "Rainbow",
          ColorFunctionScaling → False]
```

(c) Write and test a function called `nextState` which takes as its sole argument a rectangular array `mat` of 0s, 1s and 2s representing a forest with some burning trees, and returns a rectangular array representing the forest after one time step.

```
In[54]:= nextState[mat_] := Module[{
            rows = Dimensions[mat][[1]],
            columns = Dimensions[mat][[2]]},
          NewMat = ConstantArray[0, {rows, columns}];
          For[i = 1, i < rows + 1, i++,
           For[j = 1, j < columns + 1, j++,
            If[mat[[i, j]] == 1,
             If[AnyTrue[vonNeumannNeighbourhood[mat, i, j], TrueQ[# == 2] &],
              NewMat[[i, j]] = 2, NewMat[[i, j]] = 1]]]];
          NewMat]

In[55]:= MatrixPlot[nextState[mat] / 2, PlotRange → {0, 1},
          ColorFunction -> "Rainbow", ColorFunctionScaling → False]
```

(d) Write and test a function `fireStates` that takes as its sole argument a rectangular array `mat` of 0s, 1s and 2s representing a forest with some burning trees, and iterates `nextState`, starting with `mat`, until the array no longer changes, returning the list

```
{mat, nextState[mat], nextState[nextState[mat]], ...}
```

```
In[56]:= mat  = {{1, 1, 0, 0, 1, 0, 0, 1, 0, 0}, {0, 0, 1, 0, 1, 1, 1, 0, 1, 0},
            {1, 0, 1, 1, 1, 1, 1, 1, 0, 0}, {0, 0, 0, 1, 0, 0, 0, 1, 0, 1},
            {0, 0, 0, 1, 2, 0, 0, 0, 1, 0}, {1, 0, 0, 0, 1, 0, 0, 0, 0, 1},
            {0, 0, 1, 0, 0, 0, 1, 1, 1, 1}, {1, 0, 0, 1, 1, 0, 0, 0, 1, 0},
            {1, 0, 0, 0, 0, 0, 0, 1, 0, 0}, {1, 1, 0, 0, 0, 0, 1, 1, 1, 1}};

In[59]:= fireStates[mat_] := NestWhileList[nextState[#] &, mat, Unequal, All]

In[60]:= MatrixPlot[mat / 2, PlotRange → {0, 1},
          ColorFunction -> "Rainbow", ColorFunctionScaling → False]

In[61]:= MatrixPlot[Last[fireStates[mat]] / 2, PlotRange → {0, 1},
          ColorFunction -> "Rainbow", ColorFunctionScaling → False]
```

(e) Generate, and show using `ListAnimate`, a series of `MatrixPlot` frames corresponding to the spread of the forest fire starting from the following initial state:

```
In[62]:= mat = {{1, 1, 0, 0, 1, 0, 0, 1, 0, 0}, {0, 0, 1, 0, 1, 1, 1, 0, 1, 0},
            {1, 0, 1, 1, 1, 1, 1, 1, 0, 0}, {0, 0, 0, 1, 0, 0, 0, 1, 0, 1},
            {0, 0, 0, 1, 2, 0, 0, 0, 1, 0}, {1, 0, 0, 0, 1, 0, 0, 0, 0, 1},
            {0, 0, 1, 0, 0, 0, 1, 1, 1, 1}, {1, 0, 0, 1, 1, 0, 0, 0, 1, 0},
            {1, 0, 0, 0, 0, 0, 0, 1, 0, 0}, {1, 1, 0, 0, 0, 0, 1, 1, 1, 1}};

In[63]:= iterations = fireStates[mat];
         frames = Table[
            MatrixPlot[iterations[[i]] / 2, PlotRange → {0, 1}, ColorFunction → "Rainbow",
             ColorFunctionScaling → False], {i, Length[iterations]}];
         ListAnimate[
          frames]
```

(f) Write and test a function called `initialState`, which takes as its arguments a real parameter α and two dimensions `dim1` and `dim2`, and returns a `dim1` × `dim2` array of 0s, 1s and 2s.

The array should be constructed as follows. First, it should be populated randomly with 0s and 1s, with respective probabilities $1/(1+\alpha)$ and $\alpha/(1+\alpha)$. Then, exactly one square, selected at random, should be converted to a 2.

```
In[66]:= initialState[alpha_, dim1_, dim2_] := (
           NewMat = RandomChoice[{1 / (1 + alpha), alpha / (1 + alpha)} → {0, 1}, {dim1, dim2}];
           i = RandomInteger[{1, dim1}];
           j = RandomInteger[{1, dim2}];
           NewMat[[i, j]] = 2;
           NewMat)

In[67]:= initialState[.001, 8, 5] // MatrixForm

In[68]:= initialState[100, 8, 5] // MatrixForm
```

As a test

```
In[69]:= initialState[0.8, 10, 10] // MatrixForm
```

(g) What does the parameter α represent? Briefly explain.

Alpha is a parameter positively-correlated to the number density of the forest i.e. higher values of alpha mean a forest more densely packed with trees. This is clear when above we test a very small alpha (the resulting matrix has few 1s and many 0s) versus a very high alpha (which results in the opposite).

(h) Run, using `ListAnimate`, forest fire simulations for a 30 × 30 forest, generated using `initialState`, with values of α equal to 0.6, 0.8, 1.0, 1.2, 1.4, 1.6 and 1.8.

for alpha = .6

```
In[76]:= iterations = fireStates[initialState[.6, 30, 30]];
         frames = Table[
            MatrixPlot[iterations[[i]] / 2, PlotRange → {0, 1}, ColorFunction → "Rainbow",
             ColorFunctionScaling → False], {i, Length[iterations]}];
         ListAnimate[
          frames]
```

for alpha = .8

```
In[79]:= iterations = fireStates[initialState[.8, 30, 30]];
         frames = Table[
            MatrixPlot[iterations[[i]] / 2, PlotRange → {0, 1}, ColorFunction → "Rainbow",
             ColorFunctionScaling → False], {i, Length[iterations]}];
         ListAnimate[
          frames]
```

for alpha = 1

```
In[82]:= iterations = fireStates[initialState[1, 30, 30]];
         frames = Table[
            MatrixPlot[iterations[[i]] / 2, PlotRange → {0, 1}, ColorFunction → "Rainbow",
             ColorFunctionScaling → False], {i, Length[iterations]}];
         ListAnimate[
          frames]
```

for alpha = 1.2

```
In[85]:= iterations = fireStates[initialState[1.2, 30, 30]];
         frames = Table[
            MatrixPlot[iterations[[i]] / 2, PlotRange → {0, 1}, ColorFunction → "Rainbow",
             ColorFunctionScaling → False], {i, Length[iterations]}];
         ListAnimate[
          frames]
```

for alpha = 1.4

```
In[88]:= iterations = fireStates[initialState[1.4, 30, 30]];
         frames = Table[
            MatrixPlot[iterations[[i]] / 2, PlotRange → {0, 1}, ColorFunction → "Rainbow",
             ColorFunctionScaling → False], {i, Length[iterations]}];
         ListAnimate[
          frames]
```

for alpha = 1.6

```
In[91]:= iterations = fireStates[initialState[1.6, 30, 30]];
         frames = Table[
            MatrixPlot[iterations[[i]] / 2, PlotRange → {0, 1}, ColorFunction → "Rainbow",
             ColorFunctionScaling → False], {i, Length[iterations]}];
         ListAnimate[
          frames]
```

for alpha = 1.8

```
In[94]:= iterations = fireStates[initialState[1.8, 30, 30]];
         frames = Table[
            MatrixPlot[iterations[[i]] / 2, PlotRange → {0, 1}, ColorFunction → "Rainbow",
             ColorFunctionScaling → False], {i, Length[iterations]}];
         ListAnimate[
          frames]
```

(i) It is conjectured that a lower bound for the number of trees to survive the fire is the number of trees in the initial state that are *disconnected*; that is, that have no trees in their von Neumann neighbourhood.

Write two functions, each of which takes as its argument a rectangular array `mat` of 0s, 1s and 2s representing a forest with some burning trees. The first function, `survivors`, should return the number of trees left standing after a forest fire is allowed to run from initial state `mat`. The second function, `disconnected`, should return the number of trees in `mat` that have no trees, burning or not burning, in their von Neumann neighbourhoods. Use these functions to investigate the above conjecture.

```
In[98]:= survivors[mat : {{_Integer ..} ..}] := Count[FixedPoint[nextState, mat], 1, {2}]
         survivors[mat]

In[100]:= disconnected[mat : {{_Integer ..} ..}] := Module[{
             count = 0,
             rows = Dimensions[mat][[1]],
             columns = Dimensions[mat][[2]]},
           Do[If[Total[vonNeumannNeighbourhood[mat, i, j]] == 0 && mat[[i, j]] == 1, count++],
            {i, rows}, {j, columns}];
           count]

In[101]:= Quiet[disconnected[mat]]
```

```
In[102]:= EX0 = initialState[.6, 30, 30];
          EX1 = initialState[.8, 30, 30];
          EX2 = initialState[1, 30, 30];
          EX3 = initialState[1.2, 30, 30];
          EX4 = initialState[1.4, 30, 30];
          EX5 = initialState[1.6, 30, 30];
          EX6 = initialState[1.8, 30, 30];
```

```
In[110]:= Quiet[data = {{"alpha", "survivors", "disconnect"},
              {.6, survivors[Last[fireStates[EX0]]], disconnected[EX0]},
              {.8, survivors[Last[fireStates[EX1]]], disconnected[EX1]},
              {1, survivors[Last[fireStates[EX2]]], disconnected[EX2]},
              {1.2, survivors[Last[fireStates[EX3]]], disconnected[EX3]},
              {1.4, survivors[Last[fireStates[EX4]]], disconnected[EX4]},
              {1.6, survivors[Last[fireStates[EX5]]], disconnected[EX5]},
              {1.8, survivors[Last[fireStates[EX6]]], disconnected[EX6]}}];
```

```
In[111]:= Grid[data, Alignment → Left, Spacings → {2, 1}, Frame → All,
            ItemStyle → "Text", Background → {{Gray, None}, {LightGray, None}}]
```

We can see the disconnected trees do in fact provide a lower bound for the survivors.

(j) For a 30 × 30 forest, investigate the average dependence of `survivors` and `disconnected` on the parameter α.

```
In[173]:= getData[alpha_, rows_, columns_] := (
             init = initialState[alpha, rows, columns];
             countD = disconnected[init];
             countS = survivors[Last[fireStates[init]]];
             {countD, countS}
            )
          setOutput = {0};
```

```
In[129]:= getData[.6, 30, 30]
```

```
In[175]:= getRunAvg[alpha_, runs_] := (
             RunAvg = Mean[Table[Quiet[getData[alpha, 30, 30]], runs]];
             RunAvg
            )
```

```
In[176]:= getRunAvg[.6, 10]
```

```
In[177]:= getAverages[alphaSet_, runs_] := (
             Averages = getRunAvg[#, runs] & /@ alphaSet;
             Averages
            )
```

```
In[213]:= alphaSet = {0, .25, .5, .75, 1, 1.25, 1.5, 1.75, 2, 2.25, 2.5};
          output = getAverages[alphaSet, 8];
```

```
In[215]:= ListLinePlot[{Transpose[output][[1]], Transpose[output][[2]]},
           PlotLabel → "Average Dependence of Survivors and Disconnected on Alpha",
           AxesLabel → {"Alpha", "Trees Count"}, PlotLegends → {"Disconnected", "Survivors"},
           Ticks → Automatic]
```

Not only does the number of disconnected trees in the initial state serve as a lower bound for the number of survivors, plotting the average dependence of both on the alpha parameter shows the latter converges to the former in the alpha limit. This makes sense as we would expect that for more dense forest scenarios, there is much higher likelihood that having even one neighboring tree results in a given tree burning down.

(k) Briefly critique the model, and suggest possible improvements and refinements.

The main issue I see with the model is that it does not really make sense to incorporate wrap around for the edges of a forest. The fire would never spread from one corner to the complete opposite side of the forest the way it is simulated to here. Also, there is no consideration given for natural barriers to the spread of fire, such as a river or a mountain range, which could be easily incorporated to separate groups of trees. The main determinant of spread direction in a real fire is wind, and one interesting improvement would involve dynamically adjusting the VNN reach based on some random wind pattern. For example, if the wind is blowing east, the neighborhood would lose its left neighbor and gain an additional one to the right.