Daesyworld

Ben Tatman

December 15, 2017

Introduction

Incorporating the significant changes required for our model into a preexisting Daisyworld model would have provided significant difficulties and hindered out project. In order to get around this issue, a new individual based model was developed. This document aims to layout how the program can be used, how it works, and how it can be further modified.

The model is written in the C language, as this allows for a good balance between simulation speed and code readability. The name is a portmanteau of Daisyworld and DAE - referring to the incorporation of differential allelic expression in the model.

Using the Program

Command Line Options

Daesyworld is designed to be able to be generally used without recompilation. As such, a number of options are exposed via the command line interface. These have been detailed below, along with the actual effect they have on the code.

For any of these features to work an *output/* folder must be created in the directory containing the program.

X relates to an integer value, Y to a floating point value

d - produces diploid daisies ($diploid = 1$)

h - produces haploid daisies ($diploid = 0$)

s - produces sexual daisies ($sexual = 1$)

a - produces clonal (asexual) daisies ($sexual = 0$)

e - enables or disables differential allelic expression ($edea = 1$)

mY - sets the dominance coefficient to the value (0-1) given by Y ($dominance = Y$)

cY - *Deprecated*. Enables "cheats" to be produced at frequency Y ($cheat\_freq = Y, cheat = 1$)

... In early models we were intending to introduce cheating daisies (which would be able to spontaneously flip from being black cold to black hot or white cold). The code is still here, however it hasn't been subject to the same testing as the main features.

lX - sets the simulation length ($sim_length = X$)

vX - enables verbose output ($verbose\_s = X, verbose = 1$)

... If this is enabled then the program will produce files pdXXXXX.csv and tdXXXXX.csv files at a frequency given by X. These contain the positions of each individual daisy (along with their individual colours, alleles, etc), and the temperature of each cell on the map. ... Additionally, if a sexual model is being used then this will output the reproductive crosses.

u - disables output ($runprint = 0$)

b - enables peak output ($peak\_verb = 1$)

... This outputs pdXXXXX.csv and tdXXXXX.csv files when the luminosity has reached a peak value. This is defined as where ... $luminosity[t] > luminosity[t \pm 1]$

osY - enables mixed sexuals, which reproduce sexually with frequency given by Y ($sex\_freq = Y$)

... Mixed sexuals show both sexual and clonal reproduction. A value of Y=0 is equivalent to a clonal daisy, while Y=1 is equivalent to a sexual daisy.

tX - *Deprecated*. Sets initial global temperature ($global\_temperature = X$)

rX - sets resources required for a daisy to reproduce ($resources\_for\_reproducing = X$)

gX - *Deprecated*. Enabled Macromutations to occur with X times in 10,000 ($goldschmidt\_freq = X, goldschmidt = 1$)

... Macromutations are large mutations.

imY - sets initial mutation rate to be Y ($initial\_mutation\_rate = Y$)

icY - sets initial daisy colour to be Y ($initial\_colour = Y$)

itY - sets initial temperature optimum to be Y ($initial\_t\_opt = Y$)

idX - sets initial dispersal radius to be X ($initial\_dispersal = X$)

ipX - sets initial population to be X (must be less than or equal to twice the y axis size) ($initial\_pop = X$)

ikX - sets initial number of progeny to be X ($initial\_progeny = X$)

For example

```
./gaia a d e ip100 l60000
```

Would produce an initial population of 100 clonal diploid daisies (centred around the middle row), with differential allelic expression, and would then allow this system to evolve for 60,000 time steps.

Editing the Program

Later on in this document I intend to go over each step in how the program runs, however for general simulations there are a few important parts.

Luminosity

On approximately line 690 **CHECK** there are a set of lines similar to this;

```
float psdsd = (float)60000/200000;
float qwe = 400 - (pl/36);
radiation_intensity = 1 + (psdsd * sin((3.14*(float) pl)/qwe));
if (i > 500)
  pl++;
```

This defines the equation for varying the luminosity in the model (in the code it is denoted radiation_intensity, for solar radiation intensity. This is not technically correct, as in this model it is used as a multiplicative factor so generally varies between ~0 and ~2).

In this case, the luminosity varies by;

$$radiation\_intensity = 1 + 0.3sin(\frac{\pi pl}{400 - \frac{pl}{36}})$$

However this equation can be replaced to any function you like.

An additional variation in the luminosity comes from the polar-equatorial gradient. The function *radiation_factor* produces this gradient. In the stock version of Daesyworld, this is defined as

```
float radiation_factor(float n) {
  return 0.8 + 0.4*n;
```

```
}
```

This produces a lower temperature in the polar regions, and a higher temperature in the equatorial regions. By modifying this function you can affect how the daisies distribute themselves in a changing environment.

Change Simulation Parameters

A number of decisions have been made in the code which cannot be changed by the command line.

Contribution to Temperature

For example, the contribution balance between the effect of albedo and the insulative heat on the temperature, which is set to 70% albedo and 30% insulative heat. This can be modified in the function *update_t_map(void)*, by changing the parameters on the line which appears like;

```
temperature_map[x][y] = 0.7 * albedo_temp + 0.3*(temperature_map[x][y]);
```

Daisy Grid Size

Another parameter you may wish to change is the daisy grid size. This can be adjusted by changing the defined values for LANDSCAPE_X and LANDSCAPE_Y in the start of the program.

nb: if the area of the grid is greater than 10,000 you may wish to change the value of CARRYING_CAP to allow the daisies to fill the grid

Amounts of Mutation

While from the command line you can affect the rate of mutation, you can't change the amount varied in each mutation. These can be changed around line 27-28 by modifying;

```
float mutation_deviation[7] = {0.05, 0.1, 0, 0, 0, 0.1, 0.05};
float goldschmidt_mm[7] = {0.5, 4, 0, 0, 0, 4, 0.5};
```

The values in these arrays correspond to different features of the daisies.

[0] - mutation in colour A allele

[1] - mutation in temperature A allele

[2] - mutation in dispersal

[3] - mutation in number of progeny

[**4**] - mutation in mutation rate

[**5**] - mutation in temperature B allele

[**6**] - mutation in colour B allele

Reading the Output

By default Daesyworld can produce 3 types of output file - vertical.csv, pd*.csv, and td*.csv. Here I will explain the file formats;

vertical.csv

This is a very large file and is the primary output for Daesyworld. At each time step summary data is calculated for the map and model which is outputted here, which can be easily plotted and graphed with little modification. Each row is a new time step, each column is a different property of the simulation, as summarised below;

```
1  n                                    - the time step the model is currently on
2  global_temperature                   - the average temperature over the whole map
3  (float) n_t_opta[0]/n_count[0]       - the average temperature A allele for black and grey daisies
4  (float) n_t_optb[0]/n_count[0]       - the average temperature B allele for black and grey daisies
5  (float)n_t_opta[1]/n_count[1]        - the average temperature A allele for white daisies
6  (float)n_t_optb[1]/n_count[1]        - the average temperature B allele for white daisies
7  (float)n_colour[0]/n_count[0]        - the average colour of black and grey daisies
8  (float)n_colour[1]/n_count[1]        - the average colour of white daisies
9  n_count[0]                           - the number of black and grey daisies
10 n_count[1]                           - the number of white daisies
11 min_y                                - the lowest y position
12 max_y                                - the highest y position
13 num_alive                            - the total number of daisies alive
14 (float) n_progeny[0]/n_count[0]      - the average number of progeny for black and grey daisies
15 (float) n_progeny[1]/n_count[1]      - the average number of progeny for white daisies
16 (float) n_dispersal[0]/n_count[0]    - the average dispersal for black and grey daisies
17 (float) n_dispersal[1]/n_count[1]    - the average dispersal for white daisies
18 (float) n_mutation_rate[0]/n_count[0] - the average mutation rate for black and grey daisies
19 (float) n_mutation_rate[1]/n_count[1] - the average mutation rate for white daisies
20 radiation_intensity                  - the current luminosity
```

```
21 white                                - the number of white daisies (colour > 0.55)
22 black                                - the number of black daisies (colour < \0.45)
23 grey                                 - the number of grey daisies (0.45 <= colour <= 0.55)
24 num_cheat                            - the number of cheating daisies
25 switching                            - number of daisies which have switched (ie cheated) in this time
26 sd_global_temp                       - global temperature standard deviation
27 divergence                           - average allelic divergence (not accurate - use the pd files)
28 sd_divergence                        - allelic divergence standard deviation (see 27)
29 average_t_opt                        - average optimum temperature
```

pdXXXXX.csv

These files are produced at intervals defined by the controller, and provide a more fine grained look at what the daisies are actually doing.

```
22, 8, 0.55, 26.19, 2, 24.66, 26.19, 0, 21.30, 25.50, 0
16, 34, 0.59, 32.63, 2, 29.34, 32.63, 0, 29.30, 29.50, 0
41, 30, 0.62, 26.69, 2, 24.10, 26.69, 0, 24.90, 20.90, 0
38, 31, 0.64, 25.16, 2, 25.80, 25.16, 0, 24.90, 29.40, 0
45, 31, 0.63, 25.39, 2, 24.08, 25.39, 0, 24.90, 20.80, 0
11, 22, 0.60, 27.26, 2, 24.82, 27.26, 0, 20.90, 25.80, 0
0, 12, 0.55, 27.53, 2, 24.56, 27.53, 0, 20.80, 25.50, 0
```

Each row is a different daisy, and each column is a different property;

```
1  daisies[i].pos_x,                      - The location of the daisy on the x axis
2  daisies[i].pos_y,                      - location of the daisy on the y axis
3  colour(&daisies[i]),                   - The colour of the daisy
4  daisies[i].local_te,                   - The temperature of the daisy
5  daisies[i].progeny,                    - Number of progeny
6  t_opt(&daisies[i], daisies[i].local_te), - Optimum temperature of daisy
7  temperature_map[x][y]                  - Temperature of the square on which the daisy is
8  daisies[i].cheat,                      - Is the daisy cheating?
9  daisies[i].t_opt[0],                   - Temperature A allele of daisy
10 daisies[i].t_opt[1],                   - Temperature B allele of daisy
11 daisies[i].switchs                     - Is the daisy switching alleles?
```

For example, from the first line of the table above;

```
22, 8, 0.55, 26.19, 2, 24.66, 26.19, 0, 21.30, 25.50, 0
```

We can see that at (22, 8) we have a slight white daisy (c = 0.55) in an environment which is 26.19°C. This daisy has temperature alleles of 21.30°C and 25.50°C (so divergence of 4.20°C), and is expressing an optimum of 24.66°C.

tdXXXXX.csv

```
5, 12, 28.114519
5, 13, 28.604136
5, 14, 29.810356
5, 15, 29.370066
5, 16, 25.694990
5, 17, 26.836121
5, 18, 21.880693
5, 19, 29.422749
```

The columns are as follows;

```
1 x           - x location
2 y           - y location
3 temperature - the temperature of that square
```

How the program works

Daisies are defined as follows;

```
struct Daisy {
  int pos_x, pos_y, dispersal, progeny, age, generation, living, mutation_rate, cheat, switchs, current;
  float colour[2], t_opt[2], local_te, cumulated_resources;
};
```

And a daisy map is created which consists of pointers to the daisies;

```
struct Daisy * daisy_map[LANDSCAPE_X][LANDSCAPE_Y];
```

Such that daisy_map[x][y] refers to the daisy found at (x, y).

There is a similar map of temperature;

```
float temperature_map[LANDSCAPE_X][LANDSCAPE_Y];
```

Temperature

The temperature of the system is controlled entirely by albedo effects and diffusion (ie the daisis are considered to give out no heat). This is determined using the Stefano-Boltzmann law;

$$S(1-\alpha) = 4\sigma\epsilon T^4$$

The local temperature at a point is composed of 70% solar temperature (from the above law), and 30% the insulative temperature. The insulative temperature is the temperature of that point post smoothing. Smoothing is done by averaging each point with the points surrounding it to simulate the diffusion of heat throughout the environment. The solar intensity, S, was taken to be $ 1366Wmˆ{-2} $ to match that found at the surface of the earth, and so the temperatures were normalised so that the average temperature of an unchanging environment was 25 $\circ$ C. This solar intensity was then multiplied by the luminosity. The luminosity varied linearly from 0.8 in the polar regions to 1.2 in the equatorial regions;

```
float radiation_factor(float n) {
  return 0.8 + 0.4*n;
}
```

Depending on the model used, the luminosity varies differently over time. For the models in which there was constant amplitude but changing frequency of oscillations, the following equations were used;

```
float qwe = 400 - (pl/36);
radiation_intensity = 1 + (psdsd * sin((3.14*(float) pl)/qwe));
if (i > 500)
  pl++;
```

Each time step daisies then use the Stefano-Boltzmann equation to calculate their temperature (albedo_temp), which is then combined to give the overall temperature.

```
float albedo_temp = pow((solar_intensity
 >   * radiation_factor((float) daisy->pos_y/LANDSCAPE_Y) * radiation_intensity
 >   * (1-colour(daisy)))/(4*sigmaconstant), 1/4.);
// Stefan-Boltzmann Equation to calculate the daisy temperature */
albedo_temp = albedo_temp + 25 - 234;
daisy->local_te = 0.7*albedo_temp + 0.3*(temperature_map[daisy->pos_x][daisy->pos_y]);
temperature_map[daisy->pos_x][daisy->pos_y] = daisy->local_te;
```

After each run, *update_t_map*(*void*) is run. This calculates the temperatures for empty tiles, and averages out all

of the locations to allow daisies to affect their neighbours.

```
void update_t_map(void) {
  int x, y, xi, yi, min_x, max_x, min_y, max_y;
  float b_temperature_map[LANDSCAPE_X][LANDSCAPE_Y];
  float albedo_temp;
  for (x = 0; x < LANDSCAPE_X; x++) {
    for (y = 0; y < LANDSCAPE_Y; y++) {
      if (check_pos(x, y) == 0) {
        albedo_temp = pow((solar_intensity *
>  radiation_factor((float) y/LANDSCAPE_Y) * radiation_intensity * (1-0.5))/(4*sigmaconstant), 1/4.);
        albedo_temp = albedo_temp + 25 - 234;
        temperature_map[x][y] = 0.7 * albedo_temp + 0.3*(temperature_map[x][y]);


      }


      b_temperature_map[x][y] = temperature_map[x][y];
    }
  }


  // If a position doesn't have anything growing on it we need to set its temp.


  for (x = 0; x < LANDSCAPE_X; x++) {
    for (y = 0; y < LANDSCAPE_Y; y++) {
      min_x = x - 1;
      max_x = x + 2;
      min_y = y - 1;
      max_y = y + 2;
      if (min_x < 0)
        min_x = 0;
      if (max_x > LANDSCAPE_X - 1)
        max_x = LANDSCAPE_X - 1;
      if (min_y < 0)
        min_y = 0;
      if (max_y > LANDSCAPE_Y - 1)
```

```
        max_y = LANDSCAPE_Y - 1;
      float total = 0;
      for (xi = min_x; xi < max_x; xi++) {
        for (yi = min_y; yi < max_y; yi++) {
          total += b_temperature_map[xi][yi];
        }
      }
      temperature_map[x][y] = total / ((max_y - min_y) * (max_x - min_x));
    }
  }
  return;
}
```

This allows temperature feedback between the daisy and the environment.

Growth

Each time step all daisies are looped over and the function *grow(struct Daisy * daisy)* is run on them. This function ages the daisies, checks if they have died, and allows them to accumulate resources.

```
int grow(struct Daisy * daisy) {
  assert(daisy);
```

To begin with, any daisies which are dead are cleaned up and removed from the map.

```
  if (daisy->living == 0) {
    return -1;
  }


  daisy->age++;
  if (daisy->age > age_of_death || daisy->cumulated_resources < 0 ||
  >       check_pos(daisy->pos_x, daisy->pos_y) != 1) {
    if (daisy->living == 1) {
      daisy->living = 0;
      daisy_map[daisy->pos_x][daisy->pos_y] = NULL;
      num_alive--;
    }
```

```
    return -1;
  }
```

We then use the albedo of the daisy along with the solar intensity and other properties to calculate the temperature, as mentioned in the previous section.

```
  float albedo_temp = pow((solar_intensity * radiation_factor((float) daisy->pos_y/LANDSCAPE_Y) *
                  radiation_intensity * (1-colour(daisy)))/(4*sigmaconstant), 1/4.);
  // Stefan-Boltzmann Equation to calculate the daisy temperature */
  albedo_temp = albedo_temp + 25 - 234;
  daisy->local_te = 0.7*albedo_temp + 0.3*(temperature_map[daisy->pos_x][daisy->pos_y]);
  /* 70% of the local temperature is due to the daisy, 30% is due to the thermal insulation of the
        ground */
  temperature_map[daisy->pos_x][daisy->pos_y] = daisy->local_te;
```

Finally we calculate the resource accumulation of the daisies. In our simplified Daisyworld, daisies accumulate resources to reproduce. There is no limit to how many resources can be acquired from one location, and the only limitation on the accumulation of resources is the difference between the optimum temperature and the local temperature. There is additionally code in place for a nutrient gradient, however this is not in use at the moment.

```
  float delta_resources = (5 - pow(daisy->local_te - t_opt(daisy, daisy->local_te), 2))
  >               * nutrient_gradient((float) daisy->pos_x/LANDSCAPE_X);
  daisy->cumulated_resources += delta_resources;
  return 0;
}
```

Resources are determined as;

$$resources = resources_0 + (5 - (T_{local} - T_{optimum})^2)$$

Where $T_{optimum}$ is calculated as in Allele Expression.

Allele Expression

One of the main topics we were investigating was the effect of allele expression on the daisies. To this end, Daesyworld has three types of allele expression, depending on whether they are haploid or diploid, and have DAE or not.

Haploid Expression

In Haploids only a single allele is present so this is expressed. For example, if the local temperature is $24°C$, and the allele is $20°C$ then the resource accumulation will be as follows;

$$\delta resources = 5 - (24 - 20)^2 = -11$$

Which represents a starvation state.

Diploid Expression

In the non DAE case the allele expressed is chosen randomly. This is perhaps not the best way to model this, however the alternative would've required a dominance series in the temperature alleles which would have added additional unnecessary complexity to the system - four mutating factors would've had an effect on the allele expression.

For example, if the local temperature is $24°C$, one allele is $20°C$ and the other is $25°C$ then you will end up with two possible deltas;

$$\delta resources = 5 - (24 - 20)^2 = -11$$

$$\delta resources = 5 - (24 - 25)^2 = 4$$

This gives an average value for delta resources of -3.5.

Differential Allelic Expression

In the DAE case the alleles are expressed dependent on the prevailing temperature conditions. This was done with the incorporation of a dominance coefficient, which balances the expression of the two alleles. In order to calculate the two possible optimum temperatures expressed we use the following equation;

$$T_{optimum} = hT_A + (1 - h)T_B$$

We calculate both possible optimums, and use these to calculate the deltas. For example, if we take the same case as the Diploid Expression above, and take h=0.1, we would end up with $T_{optimum}$s as follows;

$$T_{optimum} = 0.1 \cdot 20 + 0.9 \cdot 25 = 24.5°C$$

$$T_{optimum} = 0.9 \cdot 20 + 0.1 \cdot 25 = 20.5°C$$

So the expressed optimum is $24.5°C$. This gives delta resources of;

$$\delta resources = 5 - (24 - 24.5)^2 = 4.75$$

Reproduction

Two possible types of reproduction are possible in Daesyworld. These are Clonal, and Sexual. In the Clonal model, a daisy's genes are passed on to their offspring directly with mutations occurring, while in the Sexual model their alleles are randomly mixed with another daisy. The Sexual model does not simulate different genders, and assumes everyone can reproduce with everyone else.

Clonal

The reproduce function takes two arguments - the first is a pointer to the parent daisy, the second is a pointer to an array where it can place the offspring.

```
int reproduce(struct Daisy * d, struct Daisy * progenitors) {
  assert(progenitors);

  int i, p, pq, current = 0 ;
  for (i = 0; i < d->progeny; i++) { // We loop over all progeny which will be produced.
    int y_delta = rng(0, 2*d->dispersal) - d->dispersal;
    int x_delta = pow(-1, rng(0, 1)) * rng(0, (int) sqrt(pow(d->dispersal, 2) - pow(y_delta, 2)));
    float deltas[7] = {0, 0, 0, 0, 0, 0, 0};
    for (p = 0; p < 7; p++) {
      if (rng(0, 1000) < d-> mutation_rate)
        deltas[p] = pow(-1, rng(0, 1)) * mutation_deviation[p];
    } // The amount to which an allele mutates is calculated.

    // Macromutations are also programmed in.
    if (goldschmidt == 1) {
      for (pq = 0; pq < 7; pq++) {
        if (rng(0, 10000) <= goldschmidt_freq)
```

```
      deltas[pq] += pow(-1, rng(0, 1)) * goldschmidt_mm[pq];
    }
  }
```

Once we have out mutation deltas we then add these to the different alleles to produce the progeny alleles. We check that the location of the new daisy is empty and is within the grid, and if so we create the new progenitor.

```
  float new_ca = d->colour[0] + deltas[0];
  float new_cb = d->colour[1] + deltas[6];
  if (new_ca > 1)
    new_ca = 1;
  if (new_cb > 1)
    new_cb = 1;
  float new_tea = d->t_opt[0] + deltas[1];
  float new_teb = d->t_opt[1] + deltas[5];

  if (diploid == 0) {
    new_tea = new_teb;
    new_ca = new_cb;
  }

  int new_d = (int) d->dispersal + deltas[2];
  int new_p = (int) d->progeny + deltas[3];
  if (new_p > MAX_PROGENY) {
    new_p = MAX_PROGENY;
  }
  int new_m = (int) d->mutation_rate + deltas[4];
  int new_x = d->pos_x + x_delta;
  int new_y = d->pos_y + y_delta;

  if (check_pos(new_x, new_y) != 1 && new_x < LANDSCAPE_X
                      && new_x > 0 && new_y > 0 && new_y < LANDSCAPE_Y) {
    progenitors[current].pos_x = new_x;
    progenitors[current].pos_y = new_y;
    progenitors[current].t_opt[0] = (new_tea>0)?new_tea:0;
```

```
      progenitors[current].t_opt[1] = (new_teb>0)?new_teb:0;
      progenitors[current].dispersal = new_d;
      progenitors[current].progeny = (new_p > 0)?new_p:0;
      progenitors[current].age = 0;
      progenitors[current].local_te = 0;
      progenitors[current].generation = d->generation + 1;
      progenitors[current].mutation_rate = new_m;
      progenitors[current].living = 1;
      progenitors[current].switchs = 0;
      progenitors[current].cheat = (d->cheat == 1)?2:d->cheat;
      progenitors[current].cumulated_resources = d->cumulated_resources / (d->progeny + 1);
      progenitors[current].colour[0] = ((new_ca > 0)?new_ca:0);
      progenitors[current].colour[1] = ((new_cb > 0)?new_cb:0);
      filled_positions_x[filled_length] = new_x;
      filled_positions_y[filled_length] = new_y;
      daisy_map[new_x][new_y] = &progenitors[current];
      filled_length++;
      current = current + 1;
    }
  }
  return current;
}
```

These daisies are then all added to the map.

Sexual

In the sexual model we need to first find a mate. This is done using the *s_reproduce()* function.

```
int s_reproduce(struct Daisy*d, struct Daisy*progenitors) {
  assert(d);
  assert(progenitors);
  int min_x, max_x, min_y, max_y, current = 0;
  min_x = d->pos_x-1;
  max_x = d->pos_x+1;
  min_y = d->pos_y-1;
```

```
  max_y = d->pos_y+1;
  if (min_x < 0)
    min_x = 0;
  if (max_x > LANDSCAPE_X-1)
    max_x = LANDSCAPE_X-1;
  if (min_y < 0)
    min_y = 0;
  if (max_y > LANDSCAPE_Y-1)
    max_y = LANDSCAPE_Y-1;
  int x, y;
  struct Daisy * p[2];
  p[0] = d;
  int found = 0;
  //printf("%d %d\n", d->age, p[0]->age);
  for (x = min_x; x <= max_x; x++) { // We loop over all neighbouring cells and look for a mate.
    for (y = min_y; y <= max_y; y++) {
      if (x != d->pos_x && y != d->pos_y && daisy_map[x][y] != NULL) {
        if (daisy_map[x][y]->living == 1 &&
              daisy_map[x][y]->cumulated_resources > resources_for_reproducing) {
          p[1] = daisy_map[x][y];
          found = 1;
        }
      }
    }
  }
  if (found == 0) { // There is nobody nearby to mate with.
    return 0;
  }

  // The parents are now p[0] and p[1].
  current = mate(p, progenitors, 0);
  return current;
}
```

Once a mate has been found, the mate function takes over. For brevity I have not quoted the whole function here,

however it follows the exact same rules as the clonal reproductive function, only it selects the alleles randomly from the two parents.

Conclusion

All of the different functions and features mentioned here when combined produce the model we used.