MMSP is a powerful diffusion modeling tool capable of modeling complex, large, three-dimensional systems. However, its capabilities are useless if the user is unfamiliar with the functions and tools MMSP has to offer. This tutorial will introduce some of MMSP's functions through step-by-step solutions of the one dimensional diffusion couple problem according to Fick's Law. The first solution demonstrates the usage of the grid class by using a c-esque analytics scheme. In the second solution, other MMSP aspects - like the laplacian operator and boundary conditions - are introduced. Finally, the solution is expanded into two dimensions to illustrate how to use MMSP multidimensional functionality.

To start off, a quick review of Fick's law: Fick's Law states the the change in concentration of a diffusive substance changes in time proportionally with the second derivative of change in space.

$$\frac{\partial C}{\partial t} = D\nabla^2 C$$

Additionally, it is possible to discretize the law in both space and time to obtain an explicit method to predict future time steps using the a map of the current concentrations in space.

$$C_{n,m+1} = D\Delta t \frac{C_{n-1,m} - 2C_{n,m} + C_{n+1,m}}{\Delta x^2} + C_{n,m}$$

In the above expression, the n subscripts refer to a position in space and the m subscripts refer to a position in time.

The expression becomes complicated when adding dimensions as the laplacian requires derivation in each dimension individually.

The first code is called 1stDiffusion.cpp. Compile using the g++ to create an executable.

g++ -I "filepath/MMSP/include" "filename" -o "executable name"

The first thing to notice is that the include, using, and finalize commands are in the same place as in the "Hello World" script handled before. These are standard for all MMSP codes. The important MMSP commands are lines 22 and 23:

grid<1,scalar<float> > GRID(1,0,length);

A breakdown of what this line means is:

grid<# of dimensions,data type> "GRID name"(Vector quantity, start position, number of terms in first dimension);

- The vector quantity is 1 unless the vector data type is used

    - vectors in MMSP are used to store multiple data values of interest in a single node on the grid
- Start position is generally 0, but any number can be used
- When using multi-dimensions, the only required change is to add more upper and lower boundaries for each subsequent dimension
    - E.g. for 2 dimensions: *grid<2,scalar<float> > GRID(1,0,lengthx,0,lengthy)*

Also, in line 28, the commands “x0(GRID)” and “x1(GRID)” are used. MMSP reads this as the initial and final nodes in the x direction, respectively. x, y, and z commands can be used for this, as well, if the number of dimensions agrees.

Some other items of interest are:

- line 43, which defines the steady state boundary conditions
- line 45, which refers to the “stability criterion.” This criterion is defined by $[D*\Delta t/(\Delta x)^2]$
- line 48, which operates on each point in the GRID
- line 49 which defines when GRID gets updated.
- line 50, which uses the swap command. The swap command has the syntax:
    - swap(*grid to be replaced*, *grid to use to replace*)
    - In this specific example, the swap command is used to swap all points in GRID1 with all corresponding points in GRID2

```
1.  include"MMSP.hpp"
2.  using namespace MMSP;
3.
4.  int main()
5.  {
6.  // Here is where all variables are called.  In c++ all variables must be defined before
    they can be used
7.  // The first word is the data type associated with the variable. “int”=integer type.
8.  int length;
9.  int offlength;
10. int iterate;
11.
12. // We make it so that the length of the diffusion couple and the number of iterations ca
    be changed each time the code is run
13. std::cout<<"input couple length"<<std::endl;
14. std::cin>>length;
15. std::cout<<""<<std::endl;
16.
17. std::cout<<"input number of iterations"<<std::endl;
18. std::cin>>iterate;
19. std::cout<<""<<std::endl;
20.
21. //here we define some 1 dimensional grids with float variable types
22. grid<1,scalar<float> > GRID(1,0,length);
23. grid<1,scalar<float> > GRID2(1,0,length);
24. //this value is defined for looping control
25. offlength=x1(GRID)-3;
```

```
26.
27. //this creates two identical grids GRID and GRID2 that are 1 for the first half and 0 for
    the second.  These represent diffusion couples.
28. for (int x=x0(GRID); x<x1(GRID); x++)
29.         if (x<length/2) {
30.                 GRID[x]=1;
31.                 GRID2[x]=1;
32.         }
33.         else {
34.                 GRID[x]=0;
35.                 GRID2[x]=0;
36.         }
37.
38. //This step controls the number of time steps based on the user input from before
39. for (int i=0;i<iterate;i++) {
40. //Iterate through grid
41.         for (int x=x0(GRID); x<x1(GRID); x++) {
42. //Define fixed boundaries by preventing the first and last nodes of the grid from
    changing
43.                 if (x==0 || x==length-1) {
44.                 }
45. //Take one time step of the discrete Fick's Law with maximum stability criterion (.5)
46. //to keep calculations from interfering with each other, the results of computations are
    stored in GRID2, then copied back to GRID after the last computation
47.                 else {
48.                         GRID2[x]=0.5*(GRID[x-1]-2*GRID[x]+GRID[x+1])+GRID[x];
49.                         if (x>offlength) {
50.                                 swap(GRID,GRID2);
51.                         }
52.                 }
53.         }
54. }
55. //This prints the results of the grid to cout
56. for (int x=x0(GRID); x<x1(GRID); x++) {
57.         std::cout<<GRID[x]<<std::endl;
58. }
59. Finalize();
60. }
```

The second time through uses more MMSP functionality. Important changes:

- line 23: instead of GRID2, the MMSP standard name “update” is used for the second grid.
- lines 37-40: define the boundary conditions (bulkier in this case, but using other MMSP standard boundaries makes this step very useful).
    - The syntax in line 37, b0(GRID,0), can be generalized to:
        - *boundary condition side*(*grid to be applied to*, *dimension to apply the boundary to*)

        - for this specific example, b0 corresponds to the left-hand boundary, GRID is the grid that the condition is applied to, and 0 corresponds to the 1st (or x) dimension.
- line 47: by using the MMSP standard loop, the grid is updated at the correct time without any commands from the operator
- line 48: ghostswap command is used in parallel processing - it has no effect when only one processor is used, but is vital when using more than one.

```
1.  #include"MMSP.hpp"
2.  using namespace MMSP;
3.
4.  // Add in Laplacian function that is built in to MMSP
5.  // Try using a grid of vectors >> one iterated with this code, one with laplacian
6.
7.
8.  //we start the program off the same way as before, but this time we do not need the
    offset length variable
9.  int main()
10. {
11. int length;
12. int iterate;
13.
14. std::cout<<"input couple length"<<std::endl;
15. std::cin>>length;
16. std::cout<<""<<std::endl;
17.
18. std::cout<<"input number of iterations"<<std::endl;
19. std::cin>>iterate;
20. std::cout<<""<<std::endl;
21.
22. grid<1,scalar<float> > GRID(1,0,length);
23. grid<1,scalar<float> > update(1,0,length);
24.
25. for (int x=x0(GRID); x<x1(GRID); x++)
26.         if (x<length/2) {
27.                 GRID[x]=1;
28.                 update[x]=1;
29.         }
30.         else {
31.                 GRID[x]=0;
32.                 update[x]=1;
33.         }
34.
35. //now we set the boundary conditions of both grids.  By choosing the Dirichlet
    conditions, it is nearly identical to the manually set boundaries.
```

```
36. //the difference is that the first and last nodes of the grid can change, and the
    theoretical points outside the grid are fixed.
37. b0(GRID,0) = Dirichlet;
38. b1(GRID,0) = Dirichlet;
39. b0(update,0) = Dirichlet;
40. b1(update,0) = Dirichlet;
41.
42. for (int k=0; k<iterate; k++) {
43.         for (int i=0; i<nodes(GRID); i++) {
44. //we can use MMSP's definition for laplacian instead of hardcoding it.
45.                 update(i)=0.5*laplacian(GRID,i)+GRID[i];
46.         }
47.         swap(GRID,update);
48.         ghostswap(GRID);
49. };
50.
51.
52. for (int x=x0(GRID); x<x1(GRID); x++)
53.         std::cout<<GRID[x]<<std::endl;
54.
55. Finalize();
56. }
```

Now, we have a two dimensional diffusion situation (the upper left quadrant of a square is one species, the other three are another). MMSP makes adding dimensions easy, the only necessary changes are:

- line 19 & 20: change 1 to 2 when setting the number of dimensions in the grid.
- lines 38-45: boundary conditions in both dimensions must be specified

Thats it. Those are the only changes that MMSP requires to operate in multiple dimensions. Other changes include redefining how the grid is set up, but generally the grid comes from a data set, so you will not have to worry about that, and the coefficient representing the diffusion was changed to reflect how the stability criterion changes in multiple dimensions.

```
1.  #include"MMSP.hpp"
2.  using namespace MMSP;
3.
4.  //we start the program off the same way as before, but this time we do not need the
5.  offset length variable. Also, this creates a square grid.
6.  int main()
7.  {
8.  int length;
9.  int iterate;
10.
11. std::cout<<"input couple length"<<std::endl;
12. std::cin>>length;
```

```
13. std::cout<<""<<std::endl;
14.
15. std::cout<<"input number of iterations"<<std::endl;
16. std::cin>>iterate;
17. std::cout<<""<<std::endl;
18.
19. grid<2,scalar<float> > GRID(1,0,length,0,length);
20. grid<2,scalar<float> > update(1,0,length,0,length);
21.
22. //This creates a grid with 1's in the upper left quadrant, and zeroes in the remaining 3
23.
24. for (int x=x0(GRID); x<x1(GRID); x++) {
25.         for (int y=y0(GRID); y<y1(GRID); y++) {
26.                 if ((x<length/2)&&(y<length/2)) {
27.                         GRID[x][y]=1;
28.                         update[x][y]=1;
29.                 }
30.                 else {
31.                         GRID[x][y]=0;
32.                         update[x][y]=1;
33.                 }
34.         }
35. }
36. //now we set the boundary conditions of both grids.  By choosing the Dirichlet
    conditions, it is nearly identical to the manually set boudaries.
37. //the difference is that the first and last nodes of the grid can change, and the
    theoretical points outside the grid are fixed.
38. b0(GRID,0) = Dirichlet;
39. b1(GRID,0) = Dirichlet;
40. b0(GRID,1) = Dirichlet;
41. b1(GRID,1) = Dirichlet;
42. b0(update,0) = Dirichlet;
43. b1(update,0) = Dirichlet;
44. b0(update,1) = Dirichlet;
45. b1(update,1) = Dirichlet;
46.
47.
48. for (int k=0; k<iterate; k++) {
49.         for (int i=0; i<nodes(GRID); i++) {
50. //we can use MMSP's definition for laplacian instead of hardcoding it.
51.                 update(i)=0.2*laplacian(GRID,i)+GRID(i);
52.         }
53.         swap(GRID,update);
54.         ghostswap(GRID);
55. };
56.
57.
```

```
58. for (int x=x0(GRID); x<x1(GRID); x++) {
59.         for (int y=y0(GRID); y<y1(GRID); y++) {
60.         std::cout<<GRID[x][y];
61.         std::cout<<"    ";
62.         }
63.         std::cout<<std::endl;
64. }
65. Finalize();
66. }
```