Tutorial 1: Getting Started

This first tutorial will lead you through the basics of how to use this program. The second tutorial explains three ways in which you can modify the program to obtain more accurate persistence diagrams. The third tutorial shows you how to graph persistent diagrams corresponding to filtrations induced by both the sub-level sets and super-level sets of the primary function. In the fourth and last tutorial we show you how to define a new primary function. If you have *Mathematica 10* then you can follow along with the *Tutorial.nb* file.

Compiling Tutorial.cpp

Before starting this tutorial, please refer to the "Computer Requirements" section in the ReadMe file, to ensure that you have downloaded all of the requisite compilers, libraries, and software.

In the folder you have downloaded, you should have all the following source code files:

- CApproximation.h, CApproximation.cpp
- CBasicPolynomial.h, CBasicPolynomial.cpp
- CComplex.h, CComplex.cpp
- CCube.h, CCube.cpp
- CFiltration.h, CFiltration.cpp
- CNode.h, CNode.cpp
- CPolynomial.h, CPolynomial.cpp
- SFace.h
- SGeometricCube.h

Additionally, you should find the file *Tutorial.cpp*. Open up *Tutorial.cpp* in your favorite source code editor and then we may begin!

The *Tutorial.cpp* file includes all of the above mentioned **.cpp* files, which in turn include all of the other structures and libraries needed by this program. One header file we need is *…/boost/numeric/interval.hpp* from the Boost library. After you've downloaded the Boost library, place the library in a directory which your compiler searches for header files. If you are using the gcc compiler from the command line, go to the directory containing the *Tutorial.cpp* file and type:

```
g++ Tutorial.cpp -o tutorial
```

If your compiler has difficulty finding your boost library, then you will need to include a path to the directory containing your Boost library. If you are using the gcc compiler, then you may compile *Tutorial.cpp* with a directive similar to:

```
g++ -I /.../boost_1_55_0 Tutorial.cpp -o tutorial
```

where `/.../` is the file path to the directory containing your Boost library.

Fabricating your First Filtration

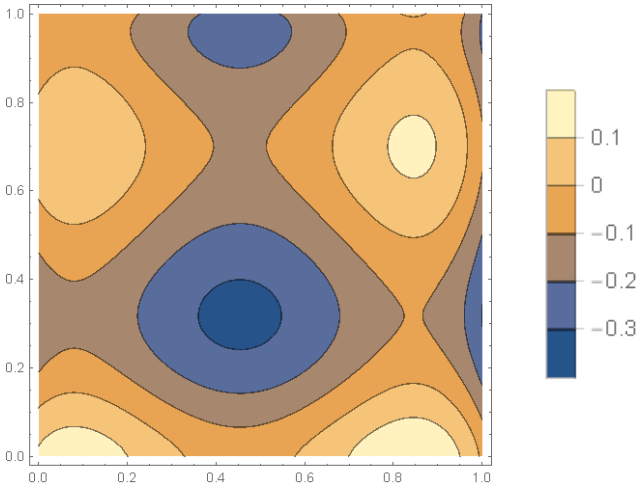
```
Verified the threshold 0.2
Verified the threshold 0.1
Verified the threshold 5.55112e-017
Failed to verify threshold -0.1  :: Exceeded maximum depth of 7.
Verified the threshold -0.2
Verified the threshold -0.3
Verified the threshold -0.4

Writing filtered CW complex to output file

Program is terminating ...
```

If you were able to compile and run the *Tutorial.cpp* file, then you should have obtained an output similar to the image to the right. Sometimes, we are not able to verify every threshold we test.
Do not despair! Even if we cannot verify every threshold, we can still construct persistence diagrams, albeit less accurate ones. In Tutorial 2, we will go over several ways reduce the number of these failures and mitigate their effect on the accuracy of our persistence diagrams.

If you obtained the output above, then you were successful in creating a filtered CW complex corresponding to the sub-level sets of the function defined below. Since the image of the function lies in the interval [-0.4,0.2] that is the range of thresholds we are interested in studying.




```
f[x,y] = 20.0*(y+0.1)(y-0.1)*(y-0.6)*(y-.8)*(y-1.05)
         -10.0*(x+0.01)*(x-0.2)*(x-0.7)*(x-0.95)
```

The primary way in which you modify and manipulate this program is through the Approximation object `approx`. In the *Tutorial.cpp* file, we called the `Verified_Filtration` member function. This uses a `for` loop to iterate through a series of equally spaced thresholds. For each threshold, it attempts to create a CW complex which is homologous to the sub/super-level set corresponding to that threshold. If successful, then that threshold is said to be verified.

More specifically, a threshold is verified if the algorithm can construct a grid where the primary function is verified on every square. A square is verified if the primary function is “well behaved” on that square. The algorithm determines that the function is “well behaved” if it passes one of several tests performed on the primary function and its partial derivatives using interval arithmetic.

If a square is not verified, then it is subdivided into 4 squares with the hope that the offspring will be verified. This process is continued until every square in a grid is verified, or a square with side length smaller than 2^{-7} (the default value) fails to be verified.

The square which caused the *Tutorial.cpp* program not to verify the threshold -0.1 had x- and y-coordinates [0.867,0.871] and [0.316,0.320]. In the diagram above, you can see that this square is located in the bottom right quadrant, where two connected components of the sub-level set are nearly touching. The primary function is actually well behaved on this square; the square's image is strictly less than -0.1002. However the error bounds produced by using interval arithmetic prevent the program from knowing this.

Calculating Persistent Homology

Using the CW approximations corresponding to each of the verified thresholds, the program constructs a single filtered CW complex. The last thing the `Verified_Filtration` function does is produce an output file called *FilteredComplex.* This file contains the information about all of the cells in the filtered CW complex: what the algebraic boundary of each cell is, and when each cell was born. (For more information about the format of this file, refer to the description of the Filtration class in the Reference Guide.)

To calculate the persistent homology of the filtration you created, you must process the *FilteredComplex* file with the Perseus software. The output file describes a regular CW complex, not a cubical complex or a simplicial complex, so you will need Version 4.1 Beta of the Perseus program.

By now you should have compiled the Perseus software into an executable file called *perseus*. To compute the persistent homology of your filtration, run the Perseus software from the command line with the following directive:

```
(path to perseus executable) cellcomp (path to input file)
```

If you use Linux or a Mac, and the *FilteredComplex* file and the *perseus* executable are in the same folder, then you would type into the command line:

```
./perseus cellcomp FilteredComplex
```

Perseus will then produce several output files. The *output_betti.txt* file should look like the following array of numbers:

```
2 2 0 0
3 4 0 0
5 1 1 0
7 1 0 0
```

You may refer to the Perseus website for how to interpret these output files, however the listed birth/death times correspond only indirectly to the actual thresholds in our filtration. In Tutorial 3 we will discuss how to process the output files from Perseus.

***Note**: If you wish to change the name of the file produced to something other than FilteredComplex, go to the file* CFiltration.cpp. *Inside the function "Export", an* `ofstream` *called "OutFile" is created. There you can set the filename corresponding to OutFile to be whatever you like.*

Tutorial 2: Improving Accuracy

Just like how a chain is only as strong as its weakest link, the precision of your persistence calculation is equal to the largest gap between verified thresholds. In the previous tutorial, the thresholds -0.4, -0.3, -0.2, 0.0, 0.1 and 0.2 were verified, and the threshold -0.1 was not verified. The largest gap between verified thresholds in this case was | -0.2 - 0.0 | = 0.2, so the persistence diagram we'd produce would be a bottleneck-distance of 0.2 away from the true persistence diagram of our primary function.

More generally, if we are computing the persistence of a function and we took for our filtration a series of thresholds with uniform spacing Δ, then the bottleneck-distance error will be equal to Δ $(F + 1)$ where *F* is the number of consecutive failures. There are two ways to reduce this error: (1) reduce the spacing between your thresholds and (2) reduce the number of consecutive failures.

***Warning**: To accurately compute a function's persistent homology, it is necessary to verify thresholds above its maximum as well as below its minimum.*

Shrinking Step Size

The first way in which we will improve the accuracy of our persistence diagrams is by verifying more thresholds. Our function only takes values in the interval [-0.4, 0.2]. We will want to only test thresholds in this range, but shrink the spacing between the thresholds down to 0.01.

To do this, in the "Tutorial.cpp" file, change the definition of `step_size` to 0.01. To have 0.2 remain the top threshold we test, the maximum number of steps we need can be calculated as (0.2 - -0.4)/ 0.01 + 1 = 61. Thereby, we should change the variable `max_steps` to equal 61.

If you then compile and run the program, about 13 thresholds will fail to be verified. Since at most two thresholds failed to be verified in a row, then the accuracy of our persistence diagram would be subject to an error of 0.03 in the bottleneck-distance.

Increasing Maximum Subdivisions

The next way in which we can improve the accuracy of our persistence diagrams is by allowing the program to use a finer resolution in constructing the discrete approximations. When the program verifies a given threshold, it subdivides areas of the domain where the super-level set

is geometrically complicated. The default setting for this program is to not allow more than 7 subdivisions of the unit square.

By increasing the maximum number of subdivisions the program allows, we can enable the program to verify thresholds which we could not verify previously. We are going to increase the maximum number of subdivisions to 15. To do this, after you've instantiated the Approximation object `approx` and before you call the `Verified_Filtration` function, enter in the line of code:

```
approx.Define_Maximum_Depth(15);
```

If you then compile and run the program, about 7 thresholds will fail to be verified. Again, two thresholds failed to be verified in a row, so the accuracy of our persistence diagram is still 0.03.

Improving Interval Precision

The last way you can shrink the error of your persistence diagrams is by improving the precision of the interval arithmetic used by the program. The program uses a very simple preprocessing technique to obtain more precise bounds for its interval arithmetic.

In general, using interval arithmetic to determine the value of a function on a large interval will produce unnecessarily large upper and lower bounds. To combat this issue, we use the fact that if we break the large interval into a bunch of tiny pieces, then the image of the large interval is equal to the union of the images of the smaller intervals.

Whenever this program uses interval arithmetic to evaluate a function on a piece of the domain, it subdivides that piece along each coordinate axis 3 times by default. Afterwards it evaluates the function on each of the constituent pieces. That means that edges get divided into 2^3 pieces and squares get divided into 4^3 pieces.

You can change the number of subdivisions the program makes by modifying the Approximation object. We are going to increase the number of subdivisions used in the interval arithmetic to 4. To do this, after you've instantiated the Approximation object `approx` and before you call the `Verified_Filtration` function, enter in the line of code:

```
approx.Define_Interval_Subdivisions(4);
```

If you then compile and run the program, then only one threshold should not have been verified. If so, then the accuracy of the persistence diagram would be subject to an error of 0.02 in the bottleneck-distance.

Warning: *Changing the interval subdivisions isn't straight forward in how it affects the program's run time. Each computation of the primary function takes longer, however larger squares may be verified, causing the resultant complexes to be smaller. Expect each increase in the number of interval subdivisions used to cause the program to run 2-4 times slower.*

Tutorial 3: Making Persistence Diagrams

By now you should be able to use this program to construct a filtered CW complex and then use the Perseus program to compute persistence intervals. We now turn our attention to graphing persistence diagrams.

Automating the Persistence Calculation

Before we get to graphing persistence diagrams, let us first automate the process for computing persistent homology. If you use Linux or a Mac, then after `Verified_Filtration` is called in the *Tutorial.cpp* file, you should enter in the following line of code:

```
system("./perseus cellcomp FilteredComplex");
```

If you use Windows, type the following:

```
system("perseus cellcomp FilteredComplex");
```

If your *perseus* executable file is not in the same folder as the *Tutorial.cpp* file then you will have to change the file path for *perseus* to the correct one.

Correcting the Persistence Intervals

A small amount of manipulation is necessary to convert the integer output of Perseus to match the real thresholds you originally told this program to compute. The formula for switching between the integer and real birth/death times depends on whether you are using sub-level sets or super-level sets for your filtration. If using **sub-level sets**, then the following formula applies:

```
Actual_Threshold = BottomThreshold +(Integer_Thresholds -1) * StepSize
```

If you are using **super-level sets** to define your filtration, then the following formula applies:

```
Actual_Threshold = TopThreshold - (Integer_Thresholds - 1) * StepSize
```

This conversion is automated by the Approximation member function `Modify_Output`. This function reads the *[output_*.txt]* files, which the program assumes are in the same folder as the *Tutorial.cpp* file. To call this function in the *Tutorial.cpp* file, after the line of code which calls the *perseus* program, enter the following line of code:

```
approx.Modify_Output(bottom_threshold, step_size, max_steps, sublevel);
```

When you run the *Tutorial.cpp* file, the `Modify_Output` function will use the above conversion formulas to produce *[real_output_*.txt]* files with the real thresholds.

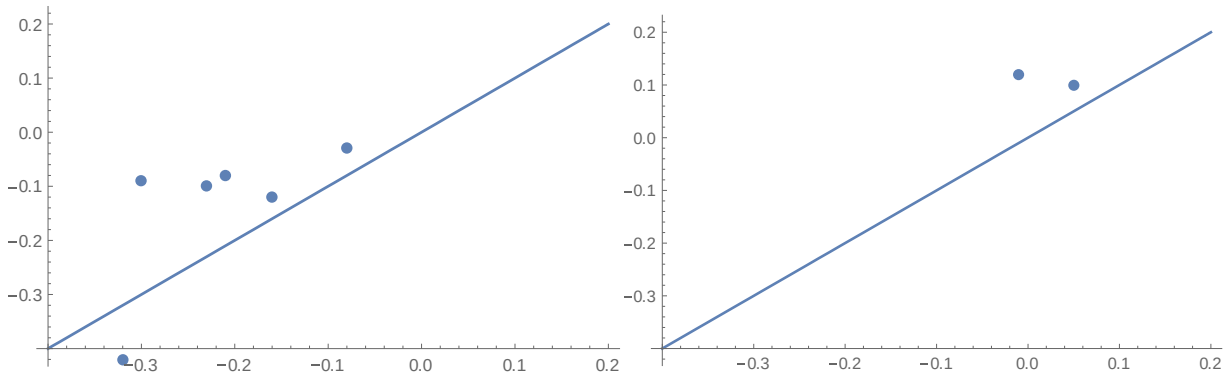
Graphing Persistence Diagrams

The process for generating persistence diagrams is largely automated in the *Tutorial.nb Mathematica* file. This file was written using version 10 of *Mathematica* and may not be entirely

compatible with earlier versions. Bundled in the Perseus software package is a MATLAB script called *Persdia* which can also take the *[real_output_*.txt]* files as input and produce persistence diagrams.

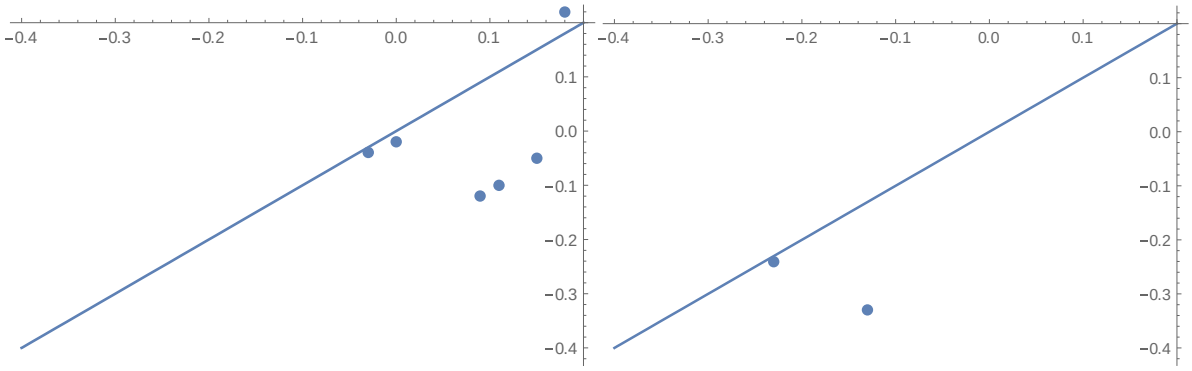
Filtrations using sub-level sets

Below are the persistence diagrams corresponding to the sub-level set filtration of our primary function. In the diagrams, the x-axis corresponds to the birth time and the y-axis corresponds to the death time. To read sub-level set persistence diagrams, you should read left-to-right and bottom-to-top.



Filtrations using super-level sets

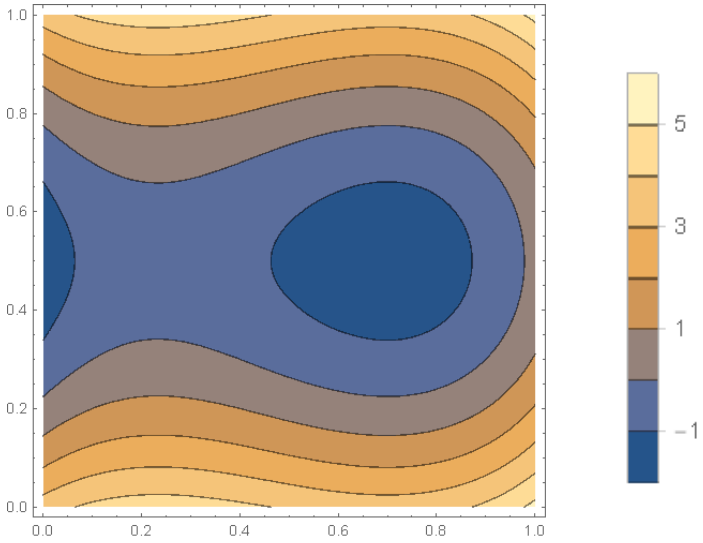
Thus far in the tutorial we have studied filtrations of our primary function induced by sub-level sets. To produce filtrations using super-level sets, in the *Tutorial.cpp* file, change the Boolean variable `sublevel` from `true` to `false`. The resulting persistence diagrams should look like the image below. Again, the x-axis corresponds to the birth time and the y-axis corresponds to the death time. To read super-level set persistence diagrams, you should read right-to-left and top-to-bottom.



Tutorial 4: Defining a New Function

With any likelihood, you did not download this program to extensively study the persistent homology of the example function I've provided. You want to study your own functions! While I cannot anticipate the function you want to study, I can show you how to define a new primary function, such as the following:

```
g(x,y):= 20.0*(y-0.3)*
(y-0.7)+20.0*(x-0.1)*
(x-0.4)*(x-0.9)
```



This program stores information about the primary functions in file *CBasicPolynomial.cpp*. Despite its name, you can choose your primary function to be something other than a polynomial.

In addition to the primary function, this program also needs to be told the partial derivatives of this function. This information is stored in the member functions `value` and `derivative`.

To define `g` as our primary function, go to the `value` function in the *CBasicPolynomial.cpp* file. There you should change how the variable `output` is defined so that it looks like this:

```
Interval output = 20.0*(y - 0.3)*(y - 0.7) + 20.0*(x - 0.1)*(x - 0.4)*(x - 0.9);
```

***Warning**: The boost interval library is strongly typed. In this program, the variables `x` and `y` are intervals of type double, and they cannot be added to / multiplied by integers.*

Now we are ready to define the partial derivatives of our primary function. For this you should go to the `derivative` function. There you should change how the variables `x_derivative` and `y_derivative` are defined so that they look like this:

```
Interval x_derivative = 20.0*((x-0.9)*(x-0.4) +(x-0.9)*(x-0.1) + (x-0.4)*(x-0.1));
Interval y_derivative = 20.0*(y-0.7) +20.0*(y-0.3);
```

***Warning:** The program does not have a safeguard against you entering the wrong derivative. If you encounter unexpected errors, one possible cause is having the wrong derivative.*