Improving a Program to Test Optics for Space-Based Telescopes

Kylie M. Wolfe and R. Steven Turley
Department of Physics and Astronomy, Brigham Young University, Provo, Utah

Abstract

Current broadband multilayer mirrors used in space-based telescopes cannot reflect light in the extreme ultraviolet range. Our research seeks to expand the wavelengths that broadband mirrors can reflect. Our research on these mirrors is conducted using a program called Octopus Measurements, which controls our system of hardware. This program had many design flaws that made it difficult to maintain, so I redesigned it to remediate those flaws.

Introduction

Our group studies multilayer mirrors for use in space-based telescopes. Currently used broadband mirrors for these telescopes cannot reflect the extreme ultraviolet (XUV) wavelengths [1, 2], so we focus on improving our mirrors' reflectance in that range of the spectrum.

Generally, the way we study our mirrors is by bouncing light off of them. We take the ratio of the intensity of the reflected light to the intensity of the light before it bounces off of the mirror to get the reflectance of the mirror at various wavelengths and angles of incidence. This occurs inside of a vacuum chamber, since our aluminum mirrors easily oxidize in air and since XUV light gets absorbed in the atmosphere.

In the simplest view of our hardware, there is a sample, plasma source, and a detector. The plasma source generates the XUV light to bounce off of the sample mirror. The detector then measures the light that bounced off of the sample, as shown in Figure 1.

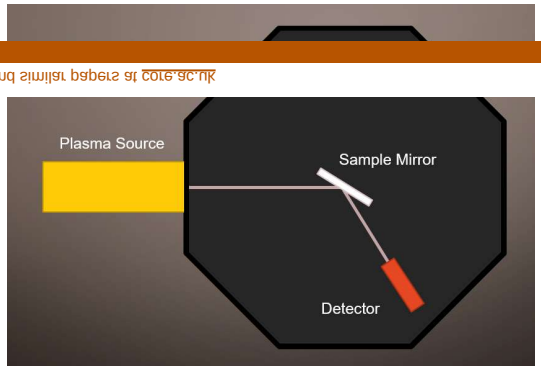

Figure 1: Diagram of hardware system

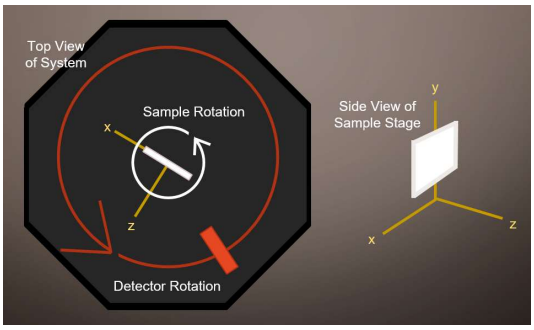

Figure 2: Available axes for a 1D scan

A simple scan across the mirror is a one-dimensional (1D) scan. The single dimension, or axis of the scan, can be chosen as the sample or detector angle, or the x, y, or z position of the sample stage, as shown in Figure 2. A user can choose an axis, which corresponds to a motor in the system, a start, stop, and increment value for the motor position, and then run a scan. The results of the scan can then be displayed and saved as amount of light recorded against each motor position.

Because all of our research must take place in a vacuum chamber, shown in Figures 1 and 2 as the large grey octagon, we can't reach in and perform these functions ourselves. Thus, we have a computer program to handle those operations for us.

The Program

The Octopus Measurements program has to be able to keep track of and control most of the hardware in the system, including but not limited to the detector, motors, pumps, USB devices, and many others. It also must provide an interface for working sure no damage will happen to the system, and perform analysis on recorded information. It is written in C# and uses the Windows Presentation Foundation (WPF) framework for its user interface (UI). Maintenance is done in Visual Studio 2013.

The program should specifically be able to count the number of photons striking our detector in a precise period of time, monitor alarm conditions on the system, monitor the pressure in parts of the system,

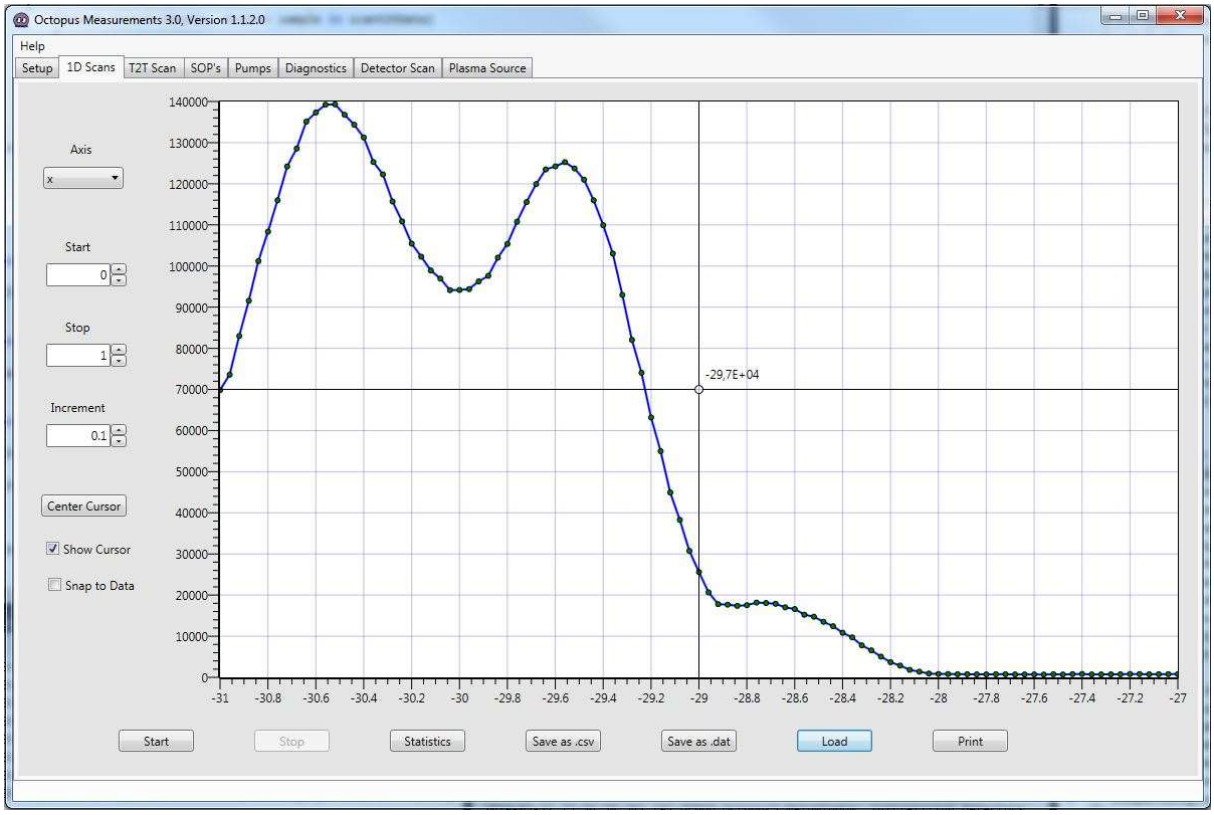

Figure 3: 1D Scans tab of Octopus Measurements user interface with data from a 1D scan

and monitor the current and voltage in our plasma source.

Another important function of the program is to capture standard operation procedures for common operations such as pumping down the system, turning on the detector, venting the chamber, and so forth. Guiding inexperienced users through the safe step-by-step procedures protects the instruments in the system.

Octopus Measurements did all of these required functions with varying success before my refactoring work, but the code underneath was difficult to work with. Most of the program's functionality was written in a single file, octopus3.xaml.cs, which contained almost 3,000 lines of code. This code ranged in function from enabling and disabling buttons to launching new threads with which to communicate with hardware. Furthermore, the user interface has eight tabs in it, each with its own functionality and purpose. This one file contained most of the code for all eight tabs.

The code was loosely organized with the #region macro, which allows the user to collapse and expand sections of code when viewed in Visual Studio. While this afforded some abstraction, regions were long and provided only cosmetic structure to the code. Though C# is fully object-oriented, almost no object-oriented design was used, and where it was used failed to take full advantage of the benefits objects can provide.

This loose organization and mixing of responsibilities made the code difficult to navigate. It was difficult to see where new code should go, and since there was no real structure to the code, it was easy to disturb the function of other code when adding new functionality or fixing bugs. Bugs were difficult to find, and when found they were often difficult to fix as well. The entire program was cognitively taxing to work with as all pieces of the functionality, from hardware to UI, were mixed in with each other.

For example, my first semester on the project I was tasked with adding controls for a few new pieces of hardware. I first built the program in a separate project, and then when it came time to integrate it in with Octopus Measurements, I was completely overwhelmed. I tried to follow the little organization that was already there, which caused me to add pieces

of my code all throughout the main octopus3 file. Then, when we ended up removing the hardware my code corresponded to, I had to hunt down all of the bits and pieces of it to remove it from the code as well.

My next project with the program was to extract code relating to certain USB devices into objects to prevent concurrent access that was crashing the program. Again, this task sent me all through the code to understand what each USB device did, and what each individual use did before I could extract its correctly.

With both of these projects, the lack of object-oriented design not only made changes hard to make but also made the program difficult to test. To truly test the program, I had to run Octopus Measurements in its entirety, manually click buttons on the user interface, and use the physical hardware. This often required help from other people, and sometimes was not even possible if the hardware could not be run.

With all of these problems, Octopus Measurements was rigid and fragile, despite generally performing its duties. It was in desperate need of refactoring, which is improving the underlying code while retaining outward functionality. My goal then was to refactor the code to make the code more readable, expandable, and otherwise maintainable.

New Architecture

The solution I created to improve Octopus Measurements is a layered architecture that breaks up the program's functionality into five high-level groups: views, presenters, services, hardware, and analysis. The dependencies between these layers can be seen in Figure 4. This architecture divides each tab's functionality not only into individual files, but also into individual responsibilities. While this division does add some overhead to the code, it is not in areas where time is critical to the function of the system. Time critical code is all in one layer, the hardware layer, where it can be optimized as much as necessary. Eventually the entire program will be broken up into these layers, but for now only the 1D Scans tab refactored into them. For simplicity's sake, I will explain the layers as though they have been fully implemented.

The view layer connects with the necessary WPF functions to make the UI work. This includes both event handlers and updates to the information on

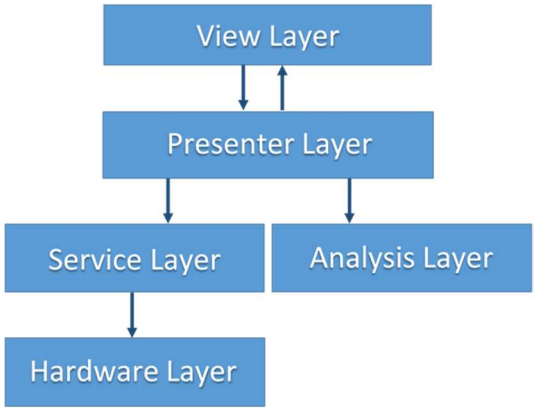

Figure 4: Layered architecture of new design. Arrows show dependencies between layers.

the screen. Since UI code is difficult to test, this layer has as little functionality in it as possible. The event handlers just pass information to the presenter layer and display popups for errors if they occur in other layers.

The presenter layer contains the functionality that was stripped from the view layer. The presenters handle any interaction by launching the appropriate services or analyses, and then returning results to the view layer to be displayed. The presenter layer also does various other tasks that don't fit cleanly into another layer, such as input validation and saving data to a file.

The analysis layer is fairly simple. It groups all of the functions that analyze collected data together into classes.

The service layer, on the other hand, is a bit complicated. It serves as the bridge between the presenter layer and the hardware layer, allowing the presenters to access hardware through a clean interface such as a 1D Scanner. Using 1D scans as an example, this provides a layer of abstraction so that the presenter can simply order a 1D scan without worrying about the details of moving motors and reading data from the detector. The OneDScanner class allows the OneDPresenter class to do a scan by providing validated scan parameters, starting a scan, and then the OneDPresenter receives the scan result data in return.

The hardware layer has a class for each type of physical hardware in the system. These classes use National Instruments libraries to communicate with the physical hardware. The hardware layer also has classes that manage the hardware to prevent multiple parts of the program from trying to access the same

hardware at the same time. This kind of access is called concurrent access, and it can cause problems such as incorrect measurements.

This new architecture takes advantage of abstractions, which allows a programmer to think only about a certain chunk of a program at a time, instead of worrying about the entire functionality all at once. It also takes advantage of the object-oriented nature of C#, which protects each piece of the program from being unintentionally or detrimentally modified by other pieces. This is called encapsulation or information hiding [3]. Using object-oriented principles also allows for automated testing of individual pieces, known as unit testing, as well as multiple pieces together, known as integration testing. Already I have been able to test this code with automated tests, instead of having to run the whole system together on the physical hardware as before.

New Hardware Management System

Most of this project consisted of refactoring the existing code but part of it included adding new functionality. The new functionality has to do with concurrent access to hardware. For example, if a 1D scan was using a motor and the detector, the scan results could be compromised if another part of the program tried to move that motor or change a setting on the detector during the scan. The big additions I made to the hardware layer prevent this kind of access from happening.

The simplest way I could think of to implement this is through a combination of the protection proxy and façade patterns, meaning a single class acts access as the restricting access point to the entire layer [4]. The problem with this design is that it would contain a method for every function that could be called on every hardware class. That would be an enormous class, and would have many of the same readability and maintainability problems that the original large octopus3.xaml.cs file had.

The solution I created instead is not perfect, but it achieves this central functionality without having one giant class to control everything. Basically, the functionality is split into three groups: an instance manager, a restrictor, and restrictables.

The instance manager is the access point for every piece of hardware. It is a singleton class, meaning only one object of it can be instantiated at a time. When it is instantiated, it also creates all of the hardware objects and keeps track of each one. Then, if any other part of the system needs to use a piece of hardware, it can only obtain it from the instance manager, which ensures duplicate hardware objects do not get created for the same piece of physical hardware.

The restrictor keeps track of what parts of the program are using which pieces of hardware at any given time. When a user class asks the restrictor for a piece of hardware, it either notifies the user class that that hardware is in use by another class, or it restricts it so that only that user class can use it. When the user class is done using the hardware, it must notify the restrictor so that other classes can access that piece of hardware.

If a class tries to use a piece of hardware without asking the restrictor for access, the hardware object itself will prevent the class from using it. That is because the hardware classes are all restrictables. They can be restricted with a key string, and if anything tries to use them without the proper key string, they will not work. The restrictor generates these key strings and restricts the hardware with them when it is granting hardware access to a user class. The restrictor then gives the proper key string to the user class.

This system has some disadvantages. For one, using a piece of hardware is complicated by the greater number of steps involved. Also, the three pieces of the system are rather tightly coupled, in that they need to know a lot of details about each other as well. This means that if one needs to change, others will likely need to change too. It also means that many of the hardware classes share some of the code that makes them restrictables. These problems are often called “code smells,” and are usually indicative of design problems [5]. Eventually these problems should be resolved, but for now the code works and succeeds in preventing unauthorized access to hardware when using this new system.

Implementation-level Refactoring

The changes to the 1D Scans tab and the underlying hardware have already gone a long way to make the program more understandable by breaking up the code into multiple files. As I did this decomposition, though, I also tried to clean up individual pieces of code to be more readable. One example of this is the code for actually performing a

```
/// <summary>
/// Called when motor is in position to start a new measurement
/// </summary>
private void scan1dMoved()
{
    log.Info("motor move completed");
    Dispatcher.Invoke(() =>
    {
        // calls to Daq tasks must be run in UI thread
        log.Info("starting detector count");
        detector.startOneCount(rate1d);
    });
}


/// <summary>
/// Called when count is complete
/// </summary>
/// <param name="counts"></param>
private void scan1dCounted(double counts)
{
    log.Info("Detector count complete");
    // Throw away the first scan.  It's usually bogus
    // question: why is this? and why is it not done for t2t scans? should be added for the t2t scan.
    if (first1d)
    {
        log.Info("First detector count, throwing away");
        first1d = false;
        DispatcherTimer timer = new DispatcherTimer();
        timer.Interval = new TimeSpan(0, 0, 2); // wait for counter to settle
        timer.Tick += scan1d_timer_Tick;
        timer.IsEnabled = true;
        return;
    }
    log.Info("Not first detector count, keeping data");
    double x = min1d + dx1d * step1d;
    Dispatcher.Invoke(() =>
    {
        scan1dData.Append(x, counts);
    });
    if (++step1d > nSteps1d || isStopped)
    {
        // I'm done.  Enable buttons and stop moving
        Dispatcher.Invoke(() =>
        {
            log.Info("Scan complete, enabling buttons and removing handlers");
            start1dButton.IsEnabled = true;
            detectorStartButton.IsEnabled = true;
            startt2tButton.IsEnabled = true;
            multiScanStartButton.IsEnabled = true;
            startSourceScan.IsEnabled = true;
            stop1dButton.IsEnabled = false;
            // Remove the completion notifiers
            detector.Completed -= scan1dCounted;
            motor1d.Notifier -= scan1dMoved;
        });
    }
    else
    {
        // I'm not done yet, move to the next point
        log.Info("Moving to next point in scan");
        step1d++;
        motor1d.Move(x + dx1d);
    }
}
```

Figure 5: The 1D scan code before it was refactored

1D scan, by moving motors and reading from the detector.

Figure 5 shows what this code looked like before the refactoring. Both of the functions in this block are callbacks, which were called when a piece of hardware finished performing its function. The first callback specifically was called when a motor finished moving to a specific position and the second was called when the detector finished reading a count. The algorithm for performing a 1D scan is exists in this code, but it is hard to see as control flows mysteriously between these two functions.

Control here starts with the first time scan1dCounted is called. On the first call, the code sits for a bit to eliminate noise at the beginning of the scan,

```
1  public async Task<Dictionary<double, double>> scanAsync()
2  {
3      await setUpScan();
4
5      log.Debug("Doing initial 1D count (thrown away)");
6      await doSingleCount(scanStartPosition);
7      if (stopFlag) return await cleanUpScan(); // user said to stop
8      await Task.Delay(2000); // wait for counter to settle
9
10     log.Info("Starting 1D scan");
11     for (double currentPosition = scanStartPosition;
12         currentPosition <= scanEndPosition;
13         currentPosition += stepIncrement)
14     {
15         if (stopFlag) break; // user said to stop
16
17         KeyValuePair<double, double> dataPoint = await doSingleCount(currentPosition);
18         notifySingleCountComplete(dataPoint.Key, dataPoint.Value);
19     }
20
21     return await cleanUpScan();
22 }
```

Figure 6: The 1D scan code after it was refactored

as shown on lines 25 to 33. If it is not the first count, then it adds the count data to the graph (line 36) and checks to see if the scan should stop (line 41). If the scan should stop, then it does the cleanup for the whole scan right from lines 46 to 55. If the scan continues, then it tells the motor to move to the next position. When the motor finishes that move, it will call the other callback, scan1dMoved which starts another detector count on line 11. This will call the scan1dCounted callback, and so forth until the scan is finished.

This code is difficult to understand as it tries to do everything from moving motors to cleaning up the UI after the scan. The way these two callbacks interact is not clear without understanding of the rest of the code. Furthermore, the code is complicated by the communication between threads. It appears that these callbacks are meant to be called from a thread other than the UI thread. Thus, each call that interacts with UI elements must do a Dispatcher.Invoke block to get back to the UI thread, as seen on lines 7, 37, and 44. These blocks add another confusing layer of complexity to this code. The comments are somewhat helpful in understanding the code, but also lead to confusion as seen in the comment conversation on lines 23 and 24.

Contrast this difficult code with the code in Figure 6, which shows this same functionality in the new, refactored version of the code. This code uses a single function, no callbacks, and async/await syntax. This makes the 1D scan algorithm much easier to see and understand. First the code prepares to scan on line 3, and then does the first count. As in the old code, it waits a moment to let the noise at the beginning of the scan settle out. This is all shown on lines 5-8. Then, the scan is achieved with a single for loop, which goes from the start position to the end position, incrementing by the specified increment value. Each time, it takes a single count measurement and processes it. Then, when the scan is done, it cleans up.

This code is essentially just the algorithm for performing a 1D scan. While this code appears significantly shorter, overall, there may be more lines of code than before. These other lines aren't pictured, but they exist in functions such as setUpScan (line 3) and doSingleCount (lines 6 and 17). Breaking the extra code into functions makes this function much easier to understand, since the reader doesn't have to worry about the details of understanding how a single count is taken, or how the scan is set up, or how any of the other functions work. The function names provide enough information to understand the algorithm without knowing the details.

The async/await paradigm is also helpful here, as no callbacks or threads have to be dealt with. The code simply sleeps until the awaited action is finished, and then it continues on to the next line. This pattern is shown in line 3, among others, where the code waits for everything to be set up before continuing on to the first count. Without async and await, this code would have to be written with a

callback, which would segment the algorithm into multiple pieces as it was segmented in the older code.

This specific example is one of many places where I applied this type of code cleanup in the 1D scan code. Generally I tried to replace callbacks with async/await functions, and I tried to ensure each function was easily understandable by breaking it up into more functions if necessary. This increased the amount of abstraction, making the functions easier to comprehend [3]. Making these kinds of implementation-level changes to the code augmented the readability improvements that breaking up the overall design into layers began. Changing the code itself to be more readable is a huge part of refactoring.

Conclusion

The changes I made to Octopus Measurements this year have made great strides in fixing some of the design issues in the original program. The 1D scan functionality has been broken up into multiple layers, the hardware is now more protected against simultaneous use by different entities in the code, and the code itself is clearer.

There is still a great more to do, however, as these changes have only been enacted in the 1D Scans tab. The next big step in the refactoring of Octopus Measurements will be applying this same architecture to the other seven tabs in the program. As I do so next year, I will be fixing some of the functionality in the program that has been broken or never fully implemented, including θ-2θ (T2T) scans. This functionality is complicated, and did not work well in the old design because of this complexity. T2T scans involve moving the sample stage to an angle θ with respect to the beam of light, and then moving the detector to an angle of twice theta to catch the light bouncing off of the mirror. The scan occurs as θ changes, moving both the sample stage and detector arm.

That functionality is relatively simple, but there are many other operations that need to be performed during a T2T scan. Occasionally during the scan, the mirror needs to be moved and the detector needs to measure how much light is coming out of the plasma source directly. This measurement is called I_0, and is used for calculating reflectance. The I_0 may vary with time, so the T2T scan code needs to take this variation into account. Furthermore, the data may need to be saved to a file periodically to preserve it in case of a crash during the scan. Additionally, the detector is unable to register the entire beam coming off of the mirror, so the detector will likely need to do a 1D scan or some other smaller scan at each θ location, and then calculate the light bouncing off of the mirror from that smaller scan. All of this extra complexity made the T2T code in the old design almost impossible to understand, which is likely why it does not yet work. This kind of scan will allow our team to more easily calculate the reflectance of our mirrors, and it will be much easier to implement using the new design.

After the whole program has been refactored and fixed, it will be much easier to add other new features, too. One of these new features could be a way to run the UI on mock hardware, thus allowing for UI testing without running on the real hardware. Another new feature that would be beneficial is a new tab that can track what pieces of hardware are currently restricted by what parts of the program, with a safety button to release restricted hardware in case a part of the program forgets to release its hardware when it is done using it.

Overall, these changes have made and will continue to make the code easier to understand, modify, fix, and expand. This will allow our team to take better measurements, as Octopus Measurements becomes flexible enough to change with the needs of the group.

References

[1] D. Allred, R. Turley, S. Thomas, S. Willett, S. Perry and M. Greenburg, "Progress towards adding EUV reflectance to broadband Al mirrors for space-based observatories", UV/Optical/IR Space Telescopes and Instruments: Innovative Technologies and Concepts VIII, vol. 10398, 2017.

[2] M. Greenburg, D. Allred and R. Turley, "Optimization of Broadband Multilayer Mirror Reflectivity via a Genetic Algorithm", The Journal of the Utah Academy of Sciences, Arts, & Letters, vol. 94, pp. 317-325, 2017 [Online]. Available: http://www.utahacademy.org/wp-content/uploads/2018/04/JUASAL-full-text-2017-revised2.pdf. [Accessed: 20- Apr- 2019]

[3] S. Nakov et al., Fundamentals of Computer Programming with C#. Veliko Tarnovo, Bulgaria: Faber, 2013, pp. 807-852. [Online]. Available: https://introprogramming.info/english-intro-csharp-

book/read-online/chapter-20-object-oriented-programming-principles/
[4] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns. Reading, Massachusetts: Addison-Wesley, 1994, pp. 185-193, 207-217.
[5] "Code Smells", Sourcemaking.com. [Online]. Available: https://sourcemaking.com/refactoring/smells. [Accessed: 20- Apr- 2019]