Supporting information for Development of the NIST X-ray Photoelectron Spectroscopy (XPS) Database, Version 5

Angela Y. Lee[1], Cedric J. Powell[2], Justin Gorham[2], Adam Morey[1], John Henry J. Scott[3] and Robert J. Hanisch[1]

[1]*Office of Data and Informatics, National Institute of Standards and Technology, Gaithersburg, MD 20899-8520*
[2]*Nano Materials Research Group, National Institute of Standards and Technology, Gaithersburg, MD 20899-8520*
[3]*Materials Measurement Science Division, National Institute of Standards and Technology, Gaithersburg, MD 20899-8520*

Table of Contents

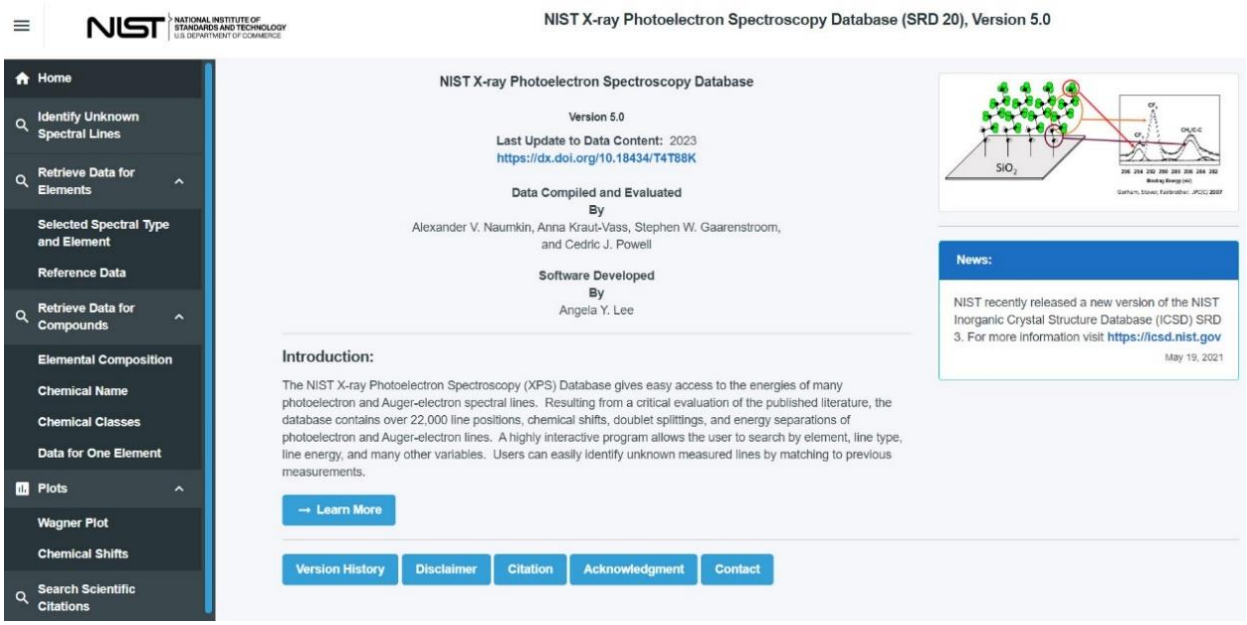

Figure S1: Home page for the NIST XPS application.

Figure S2: Results from a search for identification of an unknown spectra line.

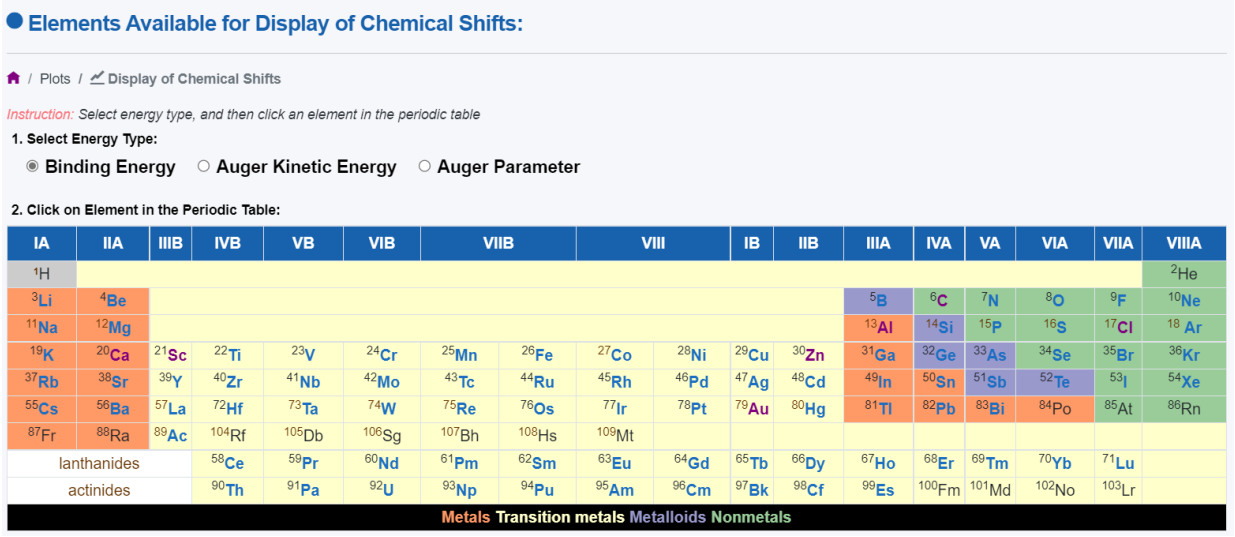

Figure S3: User interface (UI) for choose an element for a chemical shift plot.

SPECTRAL LINE
$L_3M_{45}M_{45}(^1G)$
L_3VV
$M_{23}VV$
M_3VV
$M_5N_{67}N_{67}$
N_7VV
AP-$2p_{3/2}$, $L_3M_{45}M_{45}(^1G)$
AP-$2p_{3/2}$, $L_3M_{45}M_{45}(^1G)$
AP-$4f_{7/2}$, $M_5N_{67}N_{67}$
$2p_{3/2}$
$2p_{3/2}$
$3p_{3/2}$

Figure S4: Examples of different types of formatted spectral lines. The top six entries show the format for Auger-electron lines, the next three entries starting with 'AP' show the format for Auger parameters, and the bottom three entries show the format for photoelectron lines.

Section 1: Data validator

When a user submits information for the screen shown in Fig. 3A, the input data are validated first by Blazor's built-in EditContext. The EditContext stores the form information and then validates the form components. For example, Class SearchSpectra takes the user's input from the submission. Attributes [Required] and [MaxLength] are checked by the EditContext.Validate method when the form is submitted. Messages are displayed if the validation fails. In addition, a custom validator, called InpuCustomValidator, was developed to validate a selected energy type and input energy.

Script employed

```
public class SearchSpectra {
    [Required(ErrorMessage = "Line Id is Required")]
    [MaxLength(2)]
    public string InputLineId { get; set; }

   [Required(ErrorMessage = "Energy Type is required")]
    public double? InputEnergy { get; set; }
    public double? InputError { get; set; } = 1.0;
    …
}

<EditForm EditContext="@editContext">
      <DataAnnotationsValidator />
      <ValidationSummary />
   …
  </EditForm>

bool IsValid = EditContext.Validate() && InputCustomValidato ();

public static class InputCustomValidator {
        public static bool SpectraValidator(string SpectraString, double? inpNum, double? inpErr) {
                bool RtnFlag = true;
                if (inpNum.HasValue) {
                        EnergyType SpectraType = (EnergyType)Enum.Parse(typeof(EnergyType),
SpectraString);

                        switch (SpectraType) {
                                case EnergyType.PE:
                                        if (!IsInRange(inpNum, inpErr, 2.0, 3939.9)) {
                                                RtnFlag = false;
                                        }
                                        break;
                                case EnergyType.A:
                                        if (!IsInRange(inpNum, inpErr, 20.2, 4002)) {
                                                RtnFlag = false;
                                        }
                                        break;
                                case EnergyType.AP:
                                        if (!IsInRange(inpNum, inpErr, 36.5, 5206)) {
                                                RtnFlag = false;
                                        }
                                        break;
```

```
                                        case EnergyType.DS:
                                                if (!IsInRange(inpNum, inpErr, 0.105, 126.6)) {
                                                        RtnFlag = false;
                                                }
                                                break;
                                        default:
                                                break;
                                }
                        } else {
                                RtnFlag = false;
                        }
                        return RtnFlag;
                }

                private static bool IsInRange(double? inpNum, double? inpErr, double MinVal, double MaxVal) {
                        bool RtnFlag = true;
                        if (inpNum.HasValue) {
                                double Err = 1.0;
                                if (inpErr.HasValue) {
                                        Err = inpErr.Value;
                                }
                                double ResultMin = inpNum.Value - Err;
                                double ResultMax = inpNum.Value + Err;

                                if ((ResultMin <= MinVal) || (ResultMax >= MaxVal)) {
                                        RtnFlag = false;
                                }
                        } else {
                                RtnFlag = false;
                        }
                        return RtnFlag;
                }
}

  public enum EnergyType {
     PE, A, AP, DS, CS, SICLS, INT, SA, SS, SS1, SS2, SS3, SS4, SS5, MD, CD, BD
  }
```

Section 2: Spectrum Sort Service

In atomic spectroscopy, there is a special order and format for the electron orbitals and the spectral lines observed in XPS. We built a spectrum sort service for displaying the relevant electron orbitals and spectral lines. Below are two examples of how the spectrum sort service is used.

```
LineDispList = LineDispList.OrderBy(i =>
SpectrumSortService.SortOrderList().IndexOf(i)).ToList();
LineDispiLst = LineDispList.OrderBy(i =>

SpectrumSortService.SortAPOrderList().IndexOf(i)).ToList();
```

Script employed

```
// <summary>
 /// Spectrum Sort Service
 /// </summary>
 public static class SpectrumSortService {
    public static List<string> SortOrderList() {
       List<string> SortList = new() {
          "1s", "1s, sat",
          "2s", "2p", "2p, sat", "2p1/2", "2p1/2, sat", "2p3/2", "2p3/2, sat",
          "3s", "3s, sat", "3p","3p, sat", "3p1/2","3p1/2, sat","3p3/2", "3p3/2, sat","3d", "3d3/2","3d3/2, sat", "3d5/2",
"3d5/2, sat",
          "4s", "4p", "4p1/2", "4p3/2", "4p3/2, sat", "4d", "4d, sat", "4d3/2", "4d3/2, sat", "4d5/2", "4d5/2, sat", "4f",
"4f5/2","4f5/2, sat", "4f7/2", "4f7/2, sat",
          "5s", "5s, sat","5s1/2", "5p", "5p1/2", "5p3/2", "5d", "5d3/2", "5d5/2", "5f", "5f5/2", "5f7/2",
          "6s", "6p", "6p1/2", "6p3/2"
       };
       return SortList;
    }

    public static List<string> SortAPOrderList() {
       List<string> SortList = new() {
          "AP-1s,KL23L23(1D)", "AP-1s,KVV",
          "AP-2s,KL23L23(1D)",
          "AP-2p,KL23L23(1D)", "AP-2p,L3M45M45",
          "AP-2p3/2,KL23L23(1D)", "AP-2p3/2,L23M23M23", "AP-2p3/2,L23M23V", "AP-2p3/2,L23VV", "AP-
2p3/2,L2M23M23(1D)", "AP-2p3/2,L3M23M23", "AP-2p3/2,L3M45M45(1G)", "AP-2p3/2,L3VV", "AP-
2p3/2,LVV", "AP-2p3/2,M23VV",
          "AP-3p3/2,L3M45M45(1G)", "AP-3d,L3M45M45", "AP-3d,L3M45M45(1G)", "AP-3d,M4N45N45", "AP-
3d3/2,M4N45N45",
          "AP-3d5/2,L3M45M45(1G)", "AP-3d5/2,M45N23V", "AP-3d5/2,M45N45N45", "AP-3d5/2,M4N45N45",
"AP-3d5/2,M5N23V", "AP-3d5/2,M5N45N45","AP-3d5/2,M5N45N45(1G)", "AP-3d5/2,M5N67N67", "AP-
3d5/2,M5VV",
          "AP-4d,M45N45N45",
          "AP-4d5/2,M4N45N45", "AP-4d5/2,M5N45N45","AP-4d,N45VV", "AP-4d5/2,N5O23V",
          "AP-4f7/2,M4N45N45", "AP-4f7/2,M4N67N67", "AP-4f7/2,M5N45N45", "AP-4f7/2,M5N67N67", "AP-
4f7/2,N4N67N67", "AP-4f7/2,N5N67N67", "AP-4f7/2,N67VV", "AP-4f7/2,N6O45O45"
       };
       return SortList;
    }
 }
```

Section 3: Formatting Service

A formatting service was designed and implemented to format outputs on the website. For example, MoleculeFormat and OrbitalFormat are two methods for formatting molecular formula and spectral lines from information stored as simple text format in the database.

Script employed

```
public static class FormattingService {
    public static string ChargeFormater(int charge) {
        string rtnValue = string.Empty;
        if (charge == 0) {
            return rtnValue;
        }
        rtnValue = "<sup>";
        if (charge > 0) {
            for (int i = 1; i <= charge; i++) {
                rtnValue += "+";
            }
        } else if (charge < 0) {
            for (int i = 1; i <= Math.Abs(charge); i++) {
                rtnValue += "-";
            }
        }
        rtnValue += "</sup>";
        return rtnValue;
    }

    /// <summary>
    /// Molecule Format
    /// </summary>
    /// <param name="chem"></param>
    /// <returns></returns>
    public static string MoleculeFormat(string chem) {
        if (string.IsNullOrEmpty(chem)) {
            return "";
        }
        string subStart = "<sub>";
        string subEnd = "</sub>";
        string supStart = "<sup>";
        string supEnd = "</sup>";
        string normPlus = @" + ";
        string PlusSign = @"@";
        string normMinus = @" - ";
        string myStupidMinusSign = @"$";
        bool isInSub = false;
        bool isInSup = false;

        string returned = string.Empty;
        bool chargeProcessFlag = false;
        int charge = 0;
        if (chem.Contains('-')) {
            do {
                if (chem.EndsWith("-")) {
                    charge--;
                    chem = chem[0..^1];
```

```
            chargeProcessFlag = true;
        }
    } while (chem.EndsWith("-"));
} else if (chem.Contains('+')) {
    do {
        if (chem.EndsWith("+")) {
            charge++;
            chem = chem[0..^1];
            chargeProcessFlag = true;
        }
    } while (chem.EndsWith("+"));
}
if (chargeProcessFlag) {
    if (charge > 0) {
        chem += "+";
        if (charge > 1) {
            chem += charge.ToString();
        }
    } else {
        if (charge == -1) {
            chem += "-";
        } else {
            chem += charge.ToString();
        }
    }
}
chem = chem.Replace(normPlus, PlusSign).Replace(normMinus, myStupidMinusSign);
char previousChar = ' ';
//Convert string to char array
for (int i = 0; i < chem.Length; i++) {
    switch (chem[i].ToString()) {
        case "-":
        case "+":
            if (isInSub) {
                returned += subEnd;
                isInSub = false;
            }
            returned += supStart + chem[i];
            isInSup = true;
            break;
        case "0":
        case "1":
        case "2":
        case "3":
        case "4":
        case "5":
        case "6":
        case "7":
        case "8":
        case "9":
            if (previousChar == '.' && chem.Length > i + 3 &&
                chem.Substring(i + 1, 3).Equals("H2O", StringComparison.OrdinalIgnoreCase)) {
                returned += chem[i];
                break;
            }
            if ((!isInSub) && (!isInSup)) {
```

```
                    returned += subStart + chem[i];
                    isInSub = true;
                } else {
                    returned += chem[i];
                }
                break;
            default:
                if (isInSub) {
                    returned += subEnd + chem[i];
                    isInSub = false;
                } else if (isInSup) {
                    returned += supEnd + chem[i];
                    isInSup = false;
                } else {
                    returned += chem[i];
                }
                break;
        }
        previousChar = chem[i];
    }
    if (isInSub) {
        returned += subEnd;
    } else if (isInSup) {
        returned += supEnd;
    }
    returned = returned.Replace(PlusSign, normPlus).Replace(myStupidMinusSign, normMinus);
    return returned;
}

/// <summary>
/// Orbital Format
/// </summary>
/// <param name="chem"></param>
/// <param name="InpLineId"></param>
/// <returns></returns>
public static string OrbitalFormat(string chem, string InpLineId) {
    string returned = string.Empty;
    if (string.IsNullOrEmpty(chem)) {
        return returned;
    } else {
        if (InpLineId.ToUpper().Equals(EnergyType.A.ToString()) ||
InpLineId.ToUpper().Equals(EnergyType.SA.ToString())) {
            returned = AugerOrbitalFormat(chem);
        } else if (InpLineId.ToUpper().Equals(EnergyType.AP.ToString())) {
            returned = APOrbitalFormat(chem);
        } else {
            returned = OrbitalFormat(chem);
        }
    }
    return returned;
}

/// <summary>
/// Orbital Format
/// </summary>
/// <param name="chem"></param>
```

```
/// <returns></returns>
public static string OrbitalFormat(string chem) {
    if (string.IsNullOrEmpty(chem)) {
        return string.Empty;
    }
    string returned = string.Empty;
    var chemParts = chem.Split("-");
    foreach (var item in chemParts) {
        var formattedOrbital = FormatOrbital(item);
        returned = returned + "-" + formattedOrbital;
    }
    returned = returned.TrimStart("-".ToCharArray()[0]);
    return returned;
}

/// <summary>
/// AP Orbital Formatter
/// </summary>
/// <param name="chem"></param>
/// <returns></returns>
public static string APOrbitalFormat(string chem) {
    string returned = chem;

    var chemParts = chem.Split(",");
    if (chemParts != null && chemParts.Length == 2) {
        returned = OrbitalFormat(chemParts[0]) + ", " + AugerOrbitalFormat(chemParts[1]);
    }
    return returned;

}

public static string AugerOrbitalFormat(string chem) {
    if (string.IsNullOrEmpty(chem)) {
        return "";
    }
    string subStart = "<sub>";
    string subEnd = "</sub>";
    string supStart = "<sup>";
    string supEnd = "</sup>";
    string normPlus = @" + ";
    string PlusSign = @"@";
    string normMinus = @" - ";
    string myStupidMinusSign = @"$";
    bool beginSup = false;
    bool isInSub = false;
    bool isInSup = false;

    string returned = string.Empty;

    //Convert string to char array
    for (int i = 0; i < chem.Length; i++) {
        switch (chem[i].ToString()) {
            case "(":
                beginSup = true;
                if (isInSub) {
                    returned += subEnd + chem[i];
```

```
                isInSub = false;
            } else {
                returned += chem[i];
            }
            break;
        case "0":
        case "1":
        case "2":
        case "3":
        case "4":
        case "5":
        case "6":
        case "7":
        case "8":
        case "9":
            if (beginSup) {
                returned += supStart + chem[i];
                isInSup = true;
                beginSup = false;
            } else if (!isInSub) {
                returned += subStart + chem[i];
                isInSub = true;
            } else {
                returned += chem[i];
            }
            break;
        default:
            if (isInSub) {
                returned += subEnd + chem[i];
                isInSub = false;
            } else if (isInSup) {
                returned += supEnd + chem[i];
                isInSup = false;
            } else {
                returned += chem[i];
            }
            break;
    }
  }
  if (isInSub) {
    returned += subEnd;
  } else if (isInSup) {
    returned += supEnd;
  }
  returned = returned.Replace(PlusSign, normPlus).Replace(myStupidMinusSign, normMinus);
  return returned;
}

private static string FormatOrbital(string item) {
  // looking for number / number pattern;
  if (item.Length >= 5) {
    if (Isnumeric(item[2]) && item[3].ToString() == "/" && Isnumeric(item[4])) {
      var formattedItem = ApplySubscript(item);
      return formattedItem;
    }
  }
```

```
    return item;
}

private static string ApplySubscript(string item) {
    string subStart = "<sub>";
    string subEnd = "</sub>";
    string returned = string.Empty;

    if (string.IsNullOrEmpty(item)) {
        return returned;
    }
    returned = $"{item[..2]}{subStart}{item.Substring(2, 3)}{subEnd}{item[5..]}";
    return returned;

}

private static bool Isnumeric(char v) {
    return int.TryParse(v.ToString(), out _);
}

/// <summary>
/// Decimal Formatter
/// </summary>
/// <param name="InputValue"></param>
/// <param name="numOfDigit"></param>
/// <param name="scienficExpFlag"></param>
/// <returns></returns>
public static string DecimalFormatter(double InputValue, int numOfDigit, bool scienficExpFlag) {
    string retVal;
    if (!scienficExpFlag) {
        retVal = numOfDigit switch {
            0 => string.Format("{0:#0}", InputValue),
            1 => string.Format("{0:#0.0}", InputValue),
            2 => string.Format("{0:#0.00}", InputValue),
            3 => string.Format("{0:#0.000}", InputValue),
            4 => string.Format("{0:#0.0000}", InputValue),
            5 => string.Format("{0:#0.00000}", InputValue),
            6 => string.Format("{0:#0.000000}", InputValue),
            7 => string.Format("{0:#0.0000000}", InputValue),
            8 => string.Format("{0:#0.00000000}", InputValue),
            9 => string.Format("{0:#0.000000000}", InputValue),
            _ => string.Format("{0:#0.000000}", InputValue),
        };
    } else {
        retVal = numOfDigit switch {
            0 => string.Format("{0:E0}", InputValue),
            1 => string.Format("{0:E1}", InputValue),
            2 => string.Format("{0:E2}", InputValue),
            3 => string.Format("{0:E3}", InputValue),
            4 => string.Format("{0:E4}", InputValue),
            5 => string.Format("{0:E5}", InputValue),
            6 => string.Format("{0:E6}", InputValue),
            7 => string.Format("{0:E7}", InputValue),
            8 => string.Format("{0:E8}", InputValue),
            9 => string.Format("{0:E9}", InputValue),
            _ => string.Format("{0:E10}", InputValue),
```

```
            };
        }
        return retVal;
    }
}


    /// <summary>
    /// Replace UnderScore
    /// </summary>
    /// <param name="str"></param>
    /// <returns></returns>
    public static string ReplaceUnderscore(string str) {
        if (!string.IsNullOrEmpty(str)) {
            return str.Replace('_', ' ');
        } else {
            return str;
        }
    }

    /// <summary>
    /// Check Print for Calc Shift
    /// </summary>
    /// <param name="TrueOrFalse"></param>
    /// <returns></returns>
    public static string CheckPrint4CalcShift(bool TrueOrFalse) {
        string RtnVal = string.Empty;
        if (TrueOrFalse) {
            RtnVal = "c";
        }
        return RtnVal;
    }
}
```

Section 4: Molecule Formatter

Below is an example of how the formatting service is used to display a molecular formula.

Script Employed

```
@((MarkupString)FormattingService.MoleculeFormat(Formula))

@code {
   [Parameter]
   public string Formula {get; set;} = string.Empty;
}
```

Section 5: Orbital Formatter

Orbital formatter calls formatting service to display spectral lines.

Script Employed

```
@if (!string.IsNullOrEmpty(Disp) && !string.IsNullOrEmpty(Id)) {
   @((MarkupString)FormattingService.OrbitalFormat(Disp, Id))
}
@code {
   [Parameter]
   public string Disp {get; set;} = string.Empty;

   [Parameter]
   public string Id {get; set;} = string.Empty;
}
```

Section 6: Graphical Representation of Chemical-Shift plots

We give an example of how a chemical-shift plot is generated.

Script Employed

```
public class DrawCFPlotBase : ComponentBase {
    public required Canvas2DContext _context;
    public required BECanvasComponent _canvasRef;

    [Parameter]
    public required List<ChemShiftDataPlotVM> AllWPlotChartData { get; set; }

    [Parameter]
    public string SelSpectrum { get; set; } = string.Empty;

    [Parameter]
    public string SelId { get; set; } = string.Empty;
    [Parameter]
    public bool RefreshDrawFlag { get; set; } = false;

    public List<string> SelLineDesgs { get; set; } = new();

    protected override async Task OnAfterRenderAsync(bool firstRender) {

        if (!firstRender && !RefreshDrawFlag)
            return;
        _context = await _canvasRef.CreateCanvas2DAsync();

        await _context.ClearRectAsync(0, 0, _canvasRef.Width, _canvasRef.Height);

        int chartHeight = Convert.ToInt32(_canvasRef.Height);
        int chartWidth = Convert.ToInt32(_canvasRef.Width);

        float yPlotMin = 30;
        float yPlotMax = chartHeight - yPlotMin * 4; // - 28;

        float xPlotMin = 30;
        float xPlotMax = chartWidth - xPlotMin * 6;
        //Reset the current path
        await _context.BeginPathAsync();
        await _context.SetLineWidthAsync(1);

        await _context.SetStrokeStyleAsync("#4D4E53");
        await _context.SetFontAsync("12pt Calibri");

        // Scale data points to fit on graph
        int NoPar = 0;
        int Yinc = 0;
        int elemNum = AllWPlotChartData.Count;

        double? totalMax = AllWPlotChartData.Select(s => s.EnerDispMax).Max();
        double? totalMin = AllWPlotChartData.Select(s => s.EnerDispMin).Min();

        int xDataMin = 0;
        if (totalMin.HasValue) {
```

```
            xDataMin = (int)Math.Floor(totalMin.Value * 1);
        }

        int xDataMax = 0;
        if (totalMax.HasValue) {
            xDataMax = (int)Math.Ceiling(totalMax.Value * 1);
        }

        float DistPlot = xPlotMax - xPlotMin;
        float DistData = xDataMax - xDataMin;
        if (DistData == 0) {
            var delta = 10;
            xDataMax = xDataMin + delta;
            xDataMin -= delta;
            DistData = xDataMax - xDataMin;

        }
        //else
        //{
        int MidPoint = (int)(xPlotMax + xPlotMin) / 2;

        double xPlot;
        double xData;

        foreach (ChemShiftDataPlotVM Selected in AllWPlotChartData) {
            await _context.BeginPathAsync();
            await _context.SetLineWidthAsync(1);
            Yinc = (int)yPlotMax / (elemNum + 1) * (NoPar + 1);
            await _context.SetStrokeStyleAsync(ChartColor[NoPar]);

            if (Selected != null) {
                if (Selected.EnerMedian.HasValue) {
                    xData = Selected.EnerMedian.Value;
                    xPlot = GetScaleValue(DistPlot, xPlotMin, DistData, xDataMin, xData);
                    //Mark the X
                    await _context.SetLineWidthAsync(1);
                    await _context.MoveToAsync(xPlot, Yinc + 10);
                    await _context.LineToAsync(xPlot, Yinc - 10);
                    await _context.StrokeTextAsync(FormattingService.DecimalFormatter(Selected.EnerMedian.Value,
2, false), xPlot - 20, Yinc + 25);
                }

                await _context.SetFontAsync("12pt Calibri");
                await _context.SetLineWidthAsync(1);
                if (Selected.EnerDispMin.HasValue) {
                    xData = Selected.EnerDispMin.Value;
                    xPlot = GetScaleValue(DistPlot, xPlotMin, DistData, xDataMin, xData);
                    await _context.MoveToAsync(xPlot, Yinc);
                }
                if (Selected.EnerDispMax.HasValue) {
                    xData = Selected.EnerDispMax.Value;
                    xPlot = GetScaleValue(DistPlot, xPlotMin, DistData, xDataMin, xData);
                    await _context.LineToAsync(xPlot, Yinc);
                }

                await _context.StrokeAsync();
```

```
            await _context.SetLineWidthAsync(1);
            await _context.StrokeTextAsync(Selected.FmlaStr, xPlotMax + 35, Yinc);
            NoPar++;
          }
        } // for each compound

        await _context.BeginPathAsync();

        string PlotTitle = string.Empty;
        if (!string.IsNullOrEmpty(SelId)) {
          PlotTitle = TranslationService.DecodeLineIDTitle(SelId);
        }
        await _context.SetLineWidthAsync(1);
        await _context.SetStrokeStyleAsync("#949494");
        await _context.SetFontAsync("10pt Calibri");

        //draw X axis labels
        for (int i = 0; i <= xDataMax; i++) {
          if (i + xDataMin <= xDataMax) {
            await _context.SetLineWidthAsync(1);
            //Starting point
            xPlot = DistPlot / DistData * (i + xDataMin - xDataMin) + xPlotMin;
            await _context.MoveToAsync(xPlot, yPlotMax);
            //End point
            double TickMark = yPlotMax - 10.0;
            if (i % 4 == 0) {
              TickMark = yPlotMax - 10.0;
            } else if (i % 2 == 0) {
              TickMark = yPlotMax - 10.0;
            }
            await _context.LineToAsync(xPlot, TickMark);
            await _context.StrokeAsync();
            await _context.SetStrokeStyleAsync("#000000");
            if (xDataMax - xDataMin <= 8) {
              await _context.StrokeTextAsync(Convert.ToString(xDataMin + i), xPlot - 12.0, yPlotMax + 14.0);
            } else if (i % 4 == 0) {
              await _context.StrokeTextAsync(Convert.ToString(xDataMin + i), xPlot - 12.0, yPlotMax + 14.0);
            }
            if (i == xDataMax) {
              await _context.StrokeTextAsync(Convert.ToString(xDataMin + i), xPlot - 12.0, yPlotMax + 14.0);
            }
            await DrawDividersAsync(_context, DistPlot, xPlotMin, yPlotMax, DistData, xDataMin, xDataMax, i
+ xDataMin);
          }
        }
        //Make the line visible
        //draw Y axis labels
        await _context.SetLineWidthAsync(1);
        await _context.MoveToAsync(xPlotMin, yPlotMax);
        await _context.LineToAsync(xPlotMax, yPlotMax);
        await _context.StrokeAsync();
        await _context.SetFontAsync("12pt Calibri");
        await _context.StrokeTextAsync(PlotTitle, xPlotMin + (xPlotMax - xPlotMin) / 2 - 60, yPlotMax + 30);
        await _context.SetStrokeStyleAsync("#000000");
        RefreshDrawFlag = false;
      }
```

```
}
        /// <summary>
        /// Draw Divider
        /// </summary>
        /// <param name="_context"></param>
        /// <param name="DistPlot"></param>
        /// <param name="xPlotMin"></param>
        /// <param name="yPlotMax"></param>
        /// <param name="DistData"></param>
        /// <param name="xDataMin"></param>
        /// <param name="xDataMax"></param>
        /// <param name="xCurrentData"></param>
        /// <returns></returns>
        private static async Task DrawDividersAsync(Canvas2DContext _context, float DistPlot, float xPlotMin, float
yPlotMax, float DistData, int xDataMin, int xDataMax, int xCurrentData) {
            int dividedTo = 0;
            if ((xDataMax - xDataMin) < 5)
                dividedTo = 10;
            else if ((xDataMax - xDataMin) < 10)
                dividedTo = 4;
            else if ((xDataMax - xDataMin) < 20)
                dividedTo = 2;

            if ((dividedTo == 0) || xCurrentData >= xDataMax)
                return;

            float segment = 1.00f / dividedTo;

            //draw x dividers
            for (float s = 0; s < 1; s += segment) {
                //Starting point
                float xPlot = DistPlot / DistData * (s + xCurrentData - xDataMin) + xPlotMin;
                await _context.MoveToAsync(xPlot, yPlotMax);
                //End point
                double TickMark = yPlotMax - 6.0;
                await _context.LineToAsync(xPlot, TickMark);
                await _context.StrokeAsync();
            }
            await _context.StrokeAsync();
        }
        private static double GetScaleValue(float DistPlot, float xPlotMin, float DistData, int xDataMin, double xData)
{
            double Rtn = DistPlot / DistData * (xData - xDataMin) + xPlotMin;
            return Rtn;
        }
    }
}
```