

Supplementary A

Algorithm S1: Convex hull

Convex hull algorithm is the process of mathematically calculating the points to which continuum removal will be applied in user indicated wavelength bands. It starts with extracting three points sequentially from the averaged data, selects the first two points as the reference points, and calculates the angle of the third point. If the direction moves counterclockwise, it is included as the point of the convex hull otherwise, it is removed and a new point was considered, resulting in a single shell covering all the data. The coding below generates a list named *convex hull* and calls three points from a given averaged spectral data (*x_data*, *y_data*) to (*x1*, *y1*), (*x2*, *y2*), and (*x3*, *y3*) respectively. If the outer product (*result*) of two vectors consisting of the three points is greater than zero, the middle point is added as a convex hull point. If not, the middle point is deleted from the *convex hull* list and the process of re-calculating the outer product from the existing spectral data is repeated until the relationship between the new point and the previous point is counterclockwise.

```
convexhull.Add((x_data[k], y_data[k]));
convexhull.Insert(0, (x_data[k - 1], y_data[k - 1]));
for (int i = k - 2; i >= 0; i--)
{
    if (convexhull.Count >= 2)
    {
        convexhull.Insert(0, (x_data[i], y_data[i]));
        double x1, x2, x3, y1, y2, y3, result;
        (x1, y1) = convexhull[2];
        (x2, y2) = convexhull[1];
        (x3, y3) = convexhull[0];
        result = x1 * y2 + x2 * y3 + x3 * y1 - (x2 * y1 + x3 * y2 + x1 * y3);

        if (result <= 0)
        {
            convexhull.RemoveAt(1);
            while (true)
            {
                if (convexhull.Count >= 3)
                {
                    double p1, p2, p3, q1, q2, q3;
                    (p1, q1) = convexhull[2];
                    (p2, q2) = convexhull[1];
                    (p3, q3) = convexhull[0];
                    result = p1*q2 + p2*q3 + p3*q1 - (p2*q1 + p3*q2 + p1*q3);
                    if (result >= 0) break;
                    else convexhull.RemoveAt(1);
                }
                else break;
            }
        }
    }
    else convexhull.Insert(0, (x_data[i], y_data[i]));
}
```

Algorithm S2: Gaussian deconvolution with Levenberg Marquardt method

The measured spectrum is a set of reflected values from various objects, including environmental elements within the detected range, and the target spectrum is mixed in it, so the spectrum must be decomposed to confirm the target object. A Gaussian function where each constant has a mathematical meaning is applied to deconvolute the original spectrum and the curve is fitted repeatedly using the Levenberg Marquardt (LM) method. The LM method is a nonlinear curve fitting method, which modifies the parameter to minimize the mean square error between real data and the model based on Gaussian function produced by parameters A, B, and C.

$$\text{Gaussian function: } f(x) = \frac{A}{C\sqrt{2\pi}} \exp\left(-\frac{(x-B)^2}{2C^2}\right)$$

It is crucial to select the appropriate initial values for curve fitting to be performed well, which can be determined as the first and second derivative of the raw data or added by the user clicking on the graph panel. If derivation values of the data satisfying the condition, the middle point (x_2, y_2) are added to the list box.

```

for (int j = 0; j < Data_number - 4; j++)
{
    double x, y, x1, y1, x2, y2, x3, y3, x4, y4;
    (x, y) = Data[j];
    (x1, y1) = Data[j + 1];
    (x2, y2) = Data[j + 2];
    (x3, y3) = Data[j + 3];
    (x4, y4) = Data[j + 4];

    double deriv1 = (y2 - y1) / (x2 - x1);
    double deriv2 = (y3 - y2) / (x3 - x2);
    double deriv = (deriv2 - deriv1) / (x3 - x1);
    if (deriv1 < 0.01 && deriv2 < 0.01 && deriv < 0 && y1 < y2 && y2 > y3 && y2 > 0.01 && y1 > y && y3 > y4)
    {
        if (!peak.Contains((x2, y2)))
        {
            peak.Add((x2, y2));
            peaklistbox.Items.Add((x2, y2));
        }
    }
}

```

The proportional constants have been found in the conversion to Gaussian function parameters, which allow for the proper basic expression for the curve fitting to be performed later.

```

for (int i = 0; i < peak_number; i++)
{
    parameter[3 * i] = 1.5 * y * 15;
    parameter[3 * i + 1] = x;
    parameter[3 * i + 2] = parameter[3 * i] / (y * Math.Sqrt(2 * Math.PI));
    GaussNewton(ref parameter_num, parameter, Data, ref calculated_y, ref error_square, ref mu);
}

```

GaussNewton() is a class that draws a Gaussian function using given Gaussian parameters (*parameter*). Each spectral has as many Gaussian functions as the number of different peaks in the data and the addition of all the Gaussian functions is temporarily stored in the *result* variable and then piled up in the *Calculated_y* array in this coding. The difference between

Calculated_y which is the sum of Gaussian functions and real data *y* is also represented as *error_square*.

```

private void GaussNewton()
{
    for (int i = 0; i < Data_number; i++)
    {
        double result = 0;
        (x, y) = Data[i];
        for (int j = 0; j < peak_number; j++)
        {
            A = parameter[3 * j];
            B = parameter[3 * j + 1];
            C = parameter[3 * j + 2];
            result += A / (C * Math.Sqrt(2 * Math.PI)) * Math.Exp(-(x - B) * (x - B) / (2 *
            C * C));
        }

        Calculated_y[i] = result;
        error_square += (result - y) * (result - y);
    }
}

```

In the LM method, as described in [Section 2.3](#), obtaining $\delta\mathbf{p}$ in equation (5) is implemented as coding. The residual matrix of $\mathbf{r}_i(\mathbf{p})$ is expressed as \mathbf{D} and a Jacobian matrix is replaced by \mathbf{Z} calculated prior by *GaussNewton_derive*.

To determine the convergence of the modified parameters to the original data, the gain ratio (*sigma*) was calculated every iteration so that the damping parameter (*mu*) was modified according to this value. A large value of *sigma* indicates a good approximation to modified parameters, so we can decrease *mu* so that the next LM step is reduced which means the LM method is getting closer to the Gauss-Newton method. The optimum parameters can be reached efficiently while reducing the step size carried out in this way.

```

while (iter < 50 && _error_square > max * 0.000001)
{
    GaussNewton(ref peak_number, parameter, Data, ref calculated_y, ref _error_square, ref mu);
    GaussNewton_derive(ref peak_number, parameter, Data, ref Z);

    for (int i = 0; i < Data_number; i++)
    {
        (x, y) = Data[i];
        D[i, 0] = y - calculated_y[i];
    }
    Z_T = Z.Transpose();
    ZZ_T = Z_T * Z;
    for (int row = 0; row < 3 * peak_number; row++)
    {
        for (int col = 0; col < 3 * peak_number; col++)
        {
            if (row == col)
            {
                diag[row, col] = mu * ZZ_T[row, col];
            }
        }
    }
}

```

```

deltaP= (ZZ_T + diag).Inverse() * Z_T * D;
_parameter = new double[3 * peak_number];
for (int j = 0; j < parameter_num; j++)
{
    _parameter[3*j] = parameter[3 * j] + deltaP[3 * j, 0];
    _parameter[3 * j+1] = parameter[3 * j + 1] + deltaP[3 * j + 1, 0];
    _parameter[3 * j+2] = parameter[3 * j + 2] + deltaP[3 * j + 2, 0];
}

GaussNewton(ref peak_number, _parameter, Data, ref calculated_y, ref _error_square2, ref mu);
sigma = (_error_square - _error_square2) / (1 / 2 * deltaP.Transpose() * (mu * deltaP - Z_T * D));
if(sigma[0,0]>0)
{
    parameter = _parameter;
    mu = mu * Math.Max(1 / 3, 1 - (2 * sigma[0,0] - 1) * (2 * sigma[0,0] - 1) * (2 * sigma[0,0] - 1));
    v = 1.2;
}
else
{
    mu = mu * v;
    v = 1.2 * v;
}
iter += 1;
}

```

GaussNewton_derive() is a class to build Jacobian matrix which named **Z** in this code. **Z** is formed as [Data number×peak number] matrix for each measurement and the column vectors consist of elements that are differential for the A , B , and C parameters respectively. ($J_r = \frac{\partial r_k}{\partial p_k}$, $1 \leq k \leq n$)

```

private void GaussNewton_derive()
{
    for (int i = 0; i < Data_number; i++)
    {
        (x, y) = Data[i];
        for (int j = 0; j < peak_number; j++)
        {
            A = parameter[3 * j]; B = parameter[3 * j + 1]; C = parameter[3 * j + 2];
            Z[i, 3 * j] = 1 / (C * Math.Sqrt(2 * Math.PI)) * Math.Exp(-(x - B) * (x - B) / (2 * C * C));
            Z[i, 3 * j + 1] = A / (C * Math.Sqrt(2 * Math.PI)) * Math.Exp(-(x - B * (x - B) / (2 * C * C)) * (x - B) / (C * C));
            Z[i, 3 * j + 2] = -A / (Math.Sqrt(2 * Math.PI) * C * C) * Math.Exp(-(x - B) * (x - B) / (2 * C * C)) + A
            / (C * Math.Sqrt(2 * Math.PI)) * Math.Exp(-(x - B) * (x - B) / (2 * C * C)) * (x - B) * (x - B) / (C * C);
        }
    }
}

```

Algorithm S3: Compensation for eliminating environmental effects

The spectral analysis uses an absorption spectrum observed based on electron transfer through molecular motion by light energy absorbed by atoms or molecules from the outside, and not only polymers set as target materials in this study, but also environmental conditions such as water or ice show their spectral values. If environmental spectrum peaks overlap with polymer peaks, a small number of polymer peaks may become invisible and could be misidentified. Previously, the absorption spectrum of water and ice was measured in laboratory conditions, and between 800 and 1000nm, the water peak was absorbed in 963 to 984nm and the ice at 898nm.

Compensation is performed by subtracting water or ice induced gaussian graph values from the initial deconvoluted spectrum and executing curve fitting again with subtracted data. Below is the process of conducting compensation in situations where water acts as an environmental factor. Find a Gaussian graph close to the pre-entered environmental peak (*waterpeak*), place it in the *water_gauss* array, and display the area on the graph panel showing in Fig.. After removing this element, the curve fitting is performed again using the LM method to replace the existing *parameter* array with the newly calculated *parameter* values.

```
double waterpeak = 968;
double dist2 = 1000;
int index2 = 0;
water_gauss = new double[Data_num];
for (int j = 0; j < parameter_num; j++)
{
    double tempdist2 = Math.Abs(waterpeak - parameter[3 * j + 1]);
    if (tempdist2 < dist2)
    {
        dist2 = tempdist2;
        index2 = j;
    }
}

double xx, yy;
double max = 0;
double chi2 = 10;
double v = 2;
iter = 0;
double[] new_parameter = new double[3 * peaklistbox.Items.Count];
parameter[3 * index2 + 1] = waterpeak;

for (int p = 0; p < parameter_num; p++)
{
    double A = parameter[3 * p] / (parameter[3 * p + 2] * Math.Sqrt(2 * Math.PI));
    if (A > max)
    {
        max = A;
    }
}

while (iter < 100 && chi2 > max * 0.00001)
{
    GaussNewton(ref parameter_num, ref parameter, Data, ref newcaly, ref_chi, ref_mu);
    GaussNewton_derive(ref parameter_num, ref parameter, Data, ref Z);
    for (int i = 0; i < smo_num; i++)
    {
        (xx, yy) = Data[i];
        D[i, 0] = yy - newcaly[i];
    }
}
```

```

    }

    Z_T = Z.Transpose();
    ZZ_T = Z_T * Z;

    for (int row = 0; row < 3 * parameter_num; row++)
    {
        for (int col = 0; col < 3 * parameter_num; col++)
        {
            if (row == col)
            {
                diag[row, col] = mu * 1;
            }
        }
    }
    deltaA = (ZZ_T + diag * ZZ_T).Inverse() * Z_T * D;

    for (int j = 0; j < parameter_num; j++)
    {
        if (j == index2)
        {
            double A_parameter, B_parameter, C_parameter;
            A_parameter = parameter[3 * j] + deltaA[3 * j, 0] * 0.3;
            B_parameter = parameter[3 * j + 1];
            C_parameter = parameter[3 * j + 2] + deltaA[3 * j + 2, 0] * 0.3;

            if (!(A_parameter <= 0 || C_parameter <= 0))
            {
                new_parameter[3 * j] = parameter[3 * j] + deltaA[3 * j, 0] * 0.1;
                new_parameter[3 * j + 1] = parameter[3 * j + 1];
                new_parameter[3 * j + 2] = parameter[3 * j + 2] + deltaA[3 * j + 2, 0] * 0.1;
            }
        }
        else
        {
            double A_parameter, B_parameter, C_parameter;
            A_parameter = parameter[3 * j] + deltaA[3 * j, 0] * 0.5;
            B_parameter = parameter[3 * j + 1] + deltaA[3 * j + 1, 0] * 0.5;
            C_parameter = parameter[3 * j + 2] + deltaA[3 * j + 2, 0] * 0.5;

            if (!(A_parameter <= 0 || B_parameter <= 0 || C_parameter <= 0))
            {
                new_parameter[3 * j] = parameter[3 * j] + deltaA[3 * j, 0] * 0.3;
                new_parameter[3 * j + 1] = parameter[3 * j + 1] + deltaA[3 * j + 1, 0]*0.3;
                new_parameter[3 * j + 2] = parameter[3 * j + 2] + deltaA[3 * j + 2, 0] * 0.3;
            }
        }
    }

    GaussNewton(ref parameter_num, ref new_parameter, Data, ref newcaly, ref chi2, ref mu);
    sigma = (_chi - chi2) / (1 / 2 * deltaA.Transpose() * (mu * deltaA - Z_T * D));
    if (sigma[0, 0] > 0)
    {
        parameter.Initialize();
        parameter = new_parameter;
        mu = mu * Math.Max(1 / 3, 1 - (2 * sigma[0, 0] - 1) * (2 * sigma[0, 0] - 1) * (2 * sigma[0, 0] - 1));
        v = 2;
    }
    else

```

```

{
    mu = mu * v;
    v = 2 * v;
}
iter += 1;
}

double A2, B2, C2;
A2 = parameter[3 * index2];
B2 = parameter[3 * index2 + 1];
C2 = parameter[3 * index2 + 2];

for (int i = 0; i < Data_num; i++)
{
    double x, y;
    (x, y) = Data[i];
    double gauss2 = A2 / (C2 * Math.Sqrt(2 * Math.PI)) * Math.Exp(-(x - B2) * (x - B2) / (2 * C2 * C2));
    water_gauss[i] = gauss2;
    newcaly[i] = newcaly[i] - water_gauss[i];
}

List<double> parameter_list = new List<double>();
parameter_list = parameter.ToList();
parameter_list.RemoveRange(3*index2,3);
peaklistbox.Items.RemoveAt(index2);
parameter.Initialize();
parameter = parameter_list.ToArray();

```

Algorithm S4: Polymer prediction

Plastic materials consisting of macromolecules have similar cases of absorption spectrum peak observed by similar molecular structures. In particular, in the NIR region considered in this experiment, the peaks of polymer material are mainly located between 800 to 950nm, especially when only a small amount is present, which is very difficult to distinguish.

The peak with the largest maximum of normalized reflectance (NR_{max}) value (*highpeak*) was first analyzed as the element of “PP”, “PET”, “PMMA” or “PE”, and the identification process was conducted to ensure that the polymer corresponds to the unique spectral characteristics of each of the polymer through a *switch-case*.

```

Tuple<string[], double[]> PolymerPredict(int highpeak_num,double[] cfdata)
{
    double highpeak = parameter[3 * highpeak_num + 1];
    double tempdist = 100;
    double dist = 100;
    int high_peak_idennum = 10;

    count_poly.Add("None");
    count_poly.Add("PP");
    count_poly.Add("PET");
    count_poly.Add("PMMA");
    count_poly.Add("PE");
    double[] count = Enumerable.Repeat<double>(0, count_poly.Count).ToArray<double>();

    for (int j=0;j<highest_polymer_peak.Length;j++)
    {

```

```

tempdist = Math.Abs(highpeak - highest_polymer_peak[j]);
if(tempdist<5)
{
    if(tempdist<dist)
    {
        dist = tempdist;
        high_peak_idenum = j;
    }
}

if(high_peak_idenum == 10)
{
    count[0]++;
    for (int k = 0; k < cfdata.Length / 2; k++)
    {
        Nonepeak_cal(cfdata[2 * k], ref count);
    }
}

else
{
    if(highest_polymer_name[high_peak_idenum] == "PP")
    {
        double checkpe = 10;
        if(cfdata.Length > 0)
        {
            for (int k = 0; k < cfdata.Length / 2; k++)
            {
                double checkpe_temp = Math.Abs(cfdata[2 * k] - 898);
                if(checkpe_temp < checkpe)
                {
                    checkpe = checkpe_temp;

                }
            }
        }
        if(checkpe < 3.5)
        {
            high_peak_idenum = 6;
        }
    }
}

switch (highest_polymer_name[high_peak_idenum]) {
case "PP":
    count[1] = count[1] + 1;

    if(highest_polymer_peak[high_peak_idenum] == highest_polymer_peak[1] &&
        Math.Abs(2 * Math.Sqrt(2 * Math.Log10(2)) * parameter[3 * highpeak_num + 2]) > 45)
    {
        if(cfdata.Length > 0)
        {
            for (int k = 0; k < cfdata.Length / 2; k++)
            {
                Nonepeak_cal(cfdata[2 * k], ref count);
            }
        }
        break; }
else {
```

```

        double[] parameter_peak = new double[] { 889,17, 910,13 };
        double parameter_tempparameter_least=10;
        if (cfdata.Length > 0)
        {
            for (int k = 0; k < cfdata.Length / 2; k++)
            {
                for (int l = 0; l < parameter_peak.Length / 2; l++)
                {
                    double parameter_temp = Math.Abs(cfdata[2 * k] - parameter_peak[l * 2]);
                    if (parameter_temp < parameter_tempparameter_least)
                    {
                        parameter_tempparameter_least = parameter_temp;
                    }
                }
                if (parameter_tempparameter_least < 3.5)
                {
                    count[1] = count[1] + 1;
                }
                else
                {
                    Nonepeak_cal(cfdata[2 * k], ref count);
                }
            }
            break;
        }

        case "PET":
        count[2] = count[2] + 1;
        if(highest_polymer_peak[high_peak_idenum]==highest_polymer_peak[2])
        {
            if (cfdata.Length > 0)
            {
                for (int k = 0; k < cfdata.Length / 2; k++)
                {
                    double pet_temp = Math.Abs(cfdata[2 * k] - highest_polymer_peak[3]);
                    if (pet_temp < 3.5)
                    {
                        count[2] = count[2] + 1;
                    }
                    else
                    {
                        Nonepeak_cal(cfdata[2 * k], ref count);
                    }
                }
            }
            break;
        }
        else if(highest_polymer_peak[high_peak_idenum] == highest_polymer_peak[3])
        {
            if (cfdata.Length > 0)
            {
                for (int k = 0; k < cfdata.Length / 2; k++)
                {
                    double pet_temp = Math.Abs(cfdata[2 * k] - highest_polymer_peak[2]);
                    if (pet_temp < 3.5)
                    {
                        count[2] = count[2] + 1;
                    }
                }
            }
        }
    }
}

```

```

        }
        else
        {
            Nonepeak_cal(cfdata[2 * k], ref count);
        }
    }
    break;
}
else {
    if (cfdata.Length > 0)
    {
        for (int k = 0; k < cfdata.Length / 2; k++)
        {
            Nonepeak_cal(cfdata[2 * k], ref count);
        }
    }
    break;
}

case "PMMA":
    count[3] = count[3] + 1;
    if (cfdata.Length > 0)
    {
        for (int k = 0; k < cfdata.Length / 2; k++)
        {
            Nonepeak_cal(cfdata[2 * k], ref count);
        }
    }
    break;

case "PE":
    count[4] = count[4] + 1;
    Console.WriteLine(count[4]);
    if (cfdata.Length > 0)
    {
        for (int k = 0; k < cfdata.Length / 2; k++)
        {
            double pe_temp = Math.Abs(cfdata[2 * k] - 898);
            if (pe_temp < 3.5)
            {
                count[4] = count[4] + 1;
            }
            else
            {
                Nonepeak_cal(cfdata[2 * k], ref count);
            }
        }
    }
    break;
}

return new Tuple<string[], double[]>(count_poly.ToArray(), count);
}

private void Nonepeak_cal(double peak, ref double[] count)
{
    double tempparameter_min=10;

```

```
int min_num = 10;
for(int k=0;k<polymer_peak.Length;k++)
{
    double min = Math.Abs(peak - polymer_peak[k]);
    if(tempparameter_min>min)
    {
        tempparameter_min = min;
        min_num = k;
    }
}
if(tempparameter_min<3.5)
{
    int t = count_poly.IndexOf(polymer_label[min_num]);
    count[t]++;
}
else
{
    count[0]++;
}
}
```

Figure S1: Pure polymer (PE)

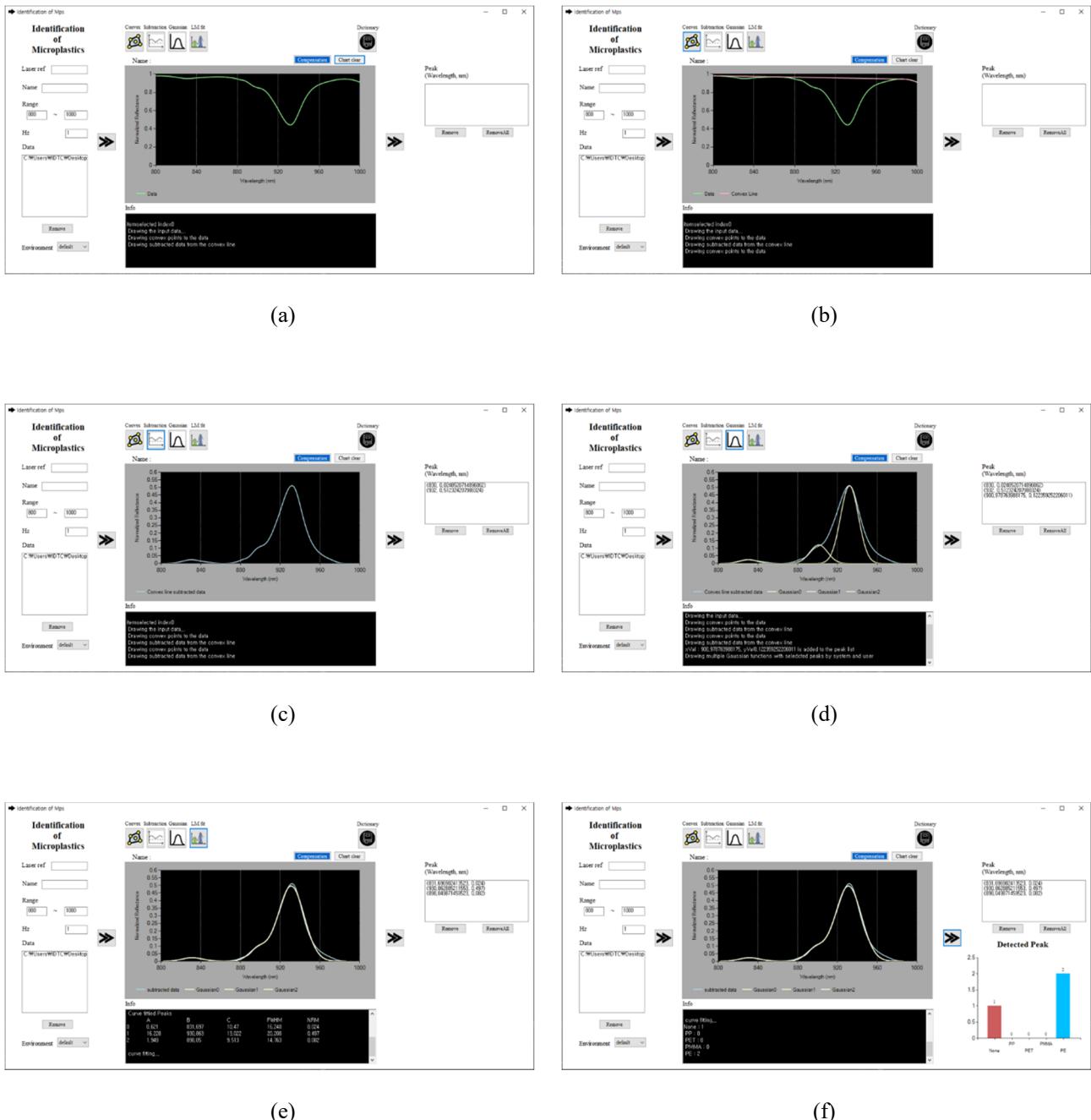


Figure S1. Spectrum analysis of PE pure polymer. (a) Draw raw data, (b) convex hull, (c) continuum line subtracted, (d) deconvoluted into multiple Gaussian graphs, (e) LM method curve fitting, and (f) identified polymer.

Figure S2: PET in water



Figure S2. Spectrum analysis of PET in water. (a) LM method curve fitting, (b) water compensated area, and re-identified result