付録 F

プログラムソースコード

1　プログラムソースコード

1.1　Program.cs

```
using Excel;
using HLFX.Block;
using HLFX.Utility;
using System;
using System.Collections.Generic;
using System.Data;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace HLFX.Sample2
{
 class Program
 {
  static void Main(string[] args)
  {

   //ログファイル作成
   using (
    var log = new System.IO.StreamWriter(@"log.csv",
    false, Encoding.GetEncoding(932)))
   {
    var simulator = new Simulator()
    {
     //計算時間、計算間隔
     StartDay = new DateTime(2015, 4, 1, 0, 0, 0),
     StopDay = new DateTime(2015, 10, 1, 0, 0, 0),
     Dt = 60,

     //ブロック設定
     Blocks = BuildBlocks(),

     //ログ出力
     Logger = (context) =>
     {
      //出力間隔の設定
      if ((context.Elasped ) % 3600 != 0) return;

      StringBuilder sb = new StringBuilder();

      sb.AppendFormat("{0},{1},", context.Current.Month
      + "月" + context.Current.Day + "日",
       context.Current.Hour + "時" +
       context.Current.Minute + "分");

      //状態出力
      foreach (var obj in context.Blocks)
      {
       if (obj is WallBlock)
       {
        //var wall = obj as WallBlock;
        //sb.AppendFormat("{0:F2}, {1:F2}, ",
        wall.TempA, wall.TempB);
       }
       else if (obj is WindowBlock)
       {
        //var wind = obj as WindowBlock;
        //sb.AppendFormat("{0:F2}, {1:F2}, ",
        wind.TempA, wind.TempB);
       }
       else if (obj is FixedConditionRoomBlock)
       {
        var room = obj as FixedConditionRoomBlock;
        sb.AppendFormat("{0:F2},", room.Temp);
       }
       else if (obj is RoomBlock)
       {
        var room = obj as RoomBlock;
        sb.AppendFormat("{0:F2},{1:F2},{2:F2},{3:F2},{
        4:F2},{5:F2},{6:F2},{7:F2},",
         room.Temp, room.MRT,room.Humidity * 1000.0,
         room.RHumidity * 100.0,
         room.SensibleHeatGenerationTotalConvection *
         3.6,
         room.VaporGenerationTotal * 1000.0 * 3600.0,
          (room.DirectSolarConvectionTotal +
         room.SkySolarConvectionTotal +
         room.ReflectSolarConvectionTotal) * 3.6,
         room.Qvent * 3.6);
       }
       else if (obj is OutsideBlock)
       {
        var _outside = obj as OutsideBlock;
        sb.AppendFormat("{0:F2}, {1:F2}, ",
        _outside.Temp, _outside.Humidity * 1000.0);
       }
       else if (obj is AirConditionerBlock)
       {
        var rac = obj as AirConditionerBlock;
        sb.AppendFormat("{0},{1:F2},{2:F2},{3:F3},{4:F
        6},", rac.Operation, rac.SensibleHeat,
        rac.LatentHeat * 2510.0 * 1000.0, rac.COP,
        rac.E_AC);
       }
       else if (obj is PersonBlock)
       {
        var person = obj as PersonBlock;
        sb.AppendFormat("{0}, ", person.InRoom);
       }
      }

      log.WriteLine(sb);

      Console.Write("*");
     }
    };

    //開始表示
    DateTime start = DateTime.Now;
    Console.WriteLine("開始:          {0}", start);
    Console.WriteLine("計算間隔(壁): {0}秒",
    simulator.Dt);
    Console.WriteLine("計算回数:      {0}回",
    simulator.GetTotalLoopCount());
    Console.WriteLine("==============================
    ==============================================
    ==========");

    //ログヘッダー用意
    log.Write("日付,時刻,");
    foreach (var obj in simulator.Blocks)
    {
     if (obj is FixedConditionRoomBlock)
      log.Write("{0},", obj.Name);
     else if (obj is RoomBlock)
```

```
      log.Write("{0},,,,,,,,", obj.Name);
    //else if (obj is WallBlock || obj is WindowBlock)

      //log.Write("{0},,", obj.Name);
    else if (obj is OutsideBlock)
      log.Write("{0},,", obj.Name);
    else if (obj is AirConditionerBlock)
      log.Write("{0},,,,,", obj.Name);
    else if (obj is PersonBlock)
      log.Write("{0},", obj.Name);
  }
  log.WriteLine("");
  log.Write(",, ");
  foreach (var obj in simulator.Blocks)
  {
   //if (obj is WallBlock || obj is WindowBlock)
   log.Write("A側温度,B側温度,");
   if (obj is FixedConditionRoomBlock) log.Write("温
   度,");
   else if (obj is RoomBlock ) log.Write("温度,MRT,絶
   対湿度,相対湿度,内部発熱対流成分[kJ/h],潜熱発生
   [g/h],日射対流成分[kJ/h],換気流入熱[kJ/h],");
   else if (obj is OutsideBlock) log.Write("温度,絶対
   湿度,");
   else if (obj is AirConditionerBlock) log.Write("運
   転,顕熱[W],潜熱[W],COP[-],消費電力[kWh],");
   else if (obj is PersonBlock) log.Write("在室,");
  }
  log.WriteLine("");

  //計算
  simulator.Run();

  //終了表示
  DateTime stop = DateTime.Now;
  Console.WriteLine("==============================
  ==============================================
  ==========");
  Console.WriteLine("終了:     {0}", stop);
  Console.WriteLine("経過時間: {0}ミリ秒", (stop -
  start).TotalMilliseconds);
  Console.WriteLine("計算速度: {0}回/秒",
  (double)simulator.GetTotalLoopCount() / (stop -
  start).Seconds);
 }
}


/// <summary>
/// ブロックの構築を行う
/// </summary>
/// <returns>構築済みのブロックのリスト</returns>
private static IList<IBlock> BuildBlocks()
{
 double inittemp = 20.0;
 double inithumidity = 7.354 / 1000.0;   ///20℃50%相
 当

 var blocks = new List<IBlock>();        // IBlockを持
 つオブジェクトのリスト

 using(FileStream stream = File.Open("input.xlsx",
 FileMode.Open, FileAccess.Read))
 using (IExcelDataReader excelReader =
 ExcelReaderFactory.CreateOpenXmlReader(stream))
 {
  excelReader.IsFirstRowAsColumnNames = true;
  DataSet data = excelReader.AsDataSet();

  #region スケジュールリスト
  var schedules = new Dictionary<string,
  ScheduleSeries>();
  if (data.Tables["スケジュール"] != null)
  {
   foreach (DataRow row in data.Tables["スケジュール
   "].Rows)
   {
    string id = Convert.ToString(row[0]);    //スケジ
    ュールID

    if (!String.IsNullOrEmpty(id))
    {
     ScheduleSeries Schedule = new ScheduleSeries();

     string sctype = Convert.ToString(row[1]);

     for (int i = 0; i < 100; i++)
     {
      int time = Convert.ToInt32(row[2 * i + 2]);
      if (time >= 2400) break;

      double scval = Convert.ToDouble(row[2 * i +
      3]);
      if (sctype == "Light" || sctype == "Equip" ||
      sctype == "Ventilation")    ///Light, Equip,
      Ventilationは[%]で入力
       scval /= 100.0;

      Schedule.UnitAdd(time, scval);
     }

     schedules.Add(id, Schedule);
    }
   }
  }

  #region 外部条件ブロック
  var outside = new OutsideBlock() { FileName =
  @"C:¥Program Files¥AE-
  CAD¥SimHeat¥weather¥6159999.sma" };  //最初に外部環
  境を生成
//キー(固定文字列)と値(outside)のセット
  blocks.Add(outside);
  #endregion


  #region  空間ブロック
  ///var rooms = new Dictionary<string, IBlock>();
  // キー(string:室名)と値(IBlock:RoomBlockのインスタ
  ンス)の組み合わせ
  var rooms = new Dictionary<string, RoomBlock>();
  foreach (DataRow row in data.Tables["居室"].Rows)
  {
   string No = Convert.ToString(row[0]);
   if (No != "")
   {
    string Name = Convert.ToString(row[1]);
    string 容積 = Convert.ToString(row[2]);
    string 種別 = Convert.ToString(row[3]);
    string 顕熱容量 = Convert.ToString(row[4]);
    ///kJ/K
    string 潜熱容量 = Convert.ToString(row[5]);
    //kg/(kg/kg')
    string 換気1 = Convert.ToString(row[6]);
```

```
    string 換気2 = Convert.ToString(row[7]);

    if (顕熱容量 == "") 顕熱容量 = "0";
    if (潜熱容量 == "") 潜熱容量 = "0";

    if (!String.IsNullOrWhiteSpace(No)
     && !String.IsNullOrWhiteSpace(Name)
     && !String.IsNullOrWhiteSpace(容積)
     && !String.IsNullOrWhiteSpace(種別))
    {
     rooms.Add(Name,
      new RoomBlock
      {
       Name = Name,
       Temp = inittemp,
       MRT = inittemp,
       Humidity = inithumidity,
       Cs = Convert.ToDouble(顕熱容量) * 1000.0,
       Cl = Convert.ToDouble(潜熱容量),
       VentNumClose = Convert.ToDouble(換気1),
       VentNumOpen = Convert.ToDouble(換気2),
       Volume = Convert.ToDouble(容積),
       TotalArea = 0.0,
       TotalFloorArea = 0.0,
       WindowOpen = 0,
       ACONOFF = 0,
       Outside = outside
      });
      //RoomBlockインスタンスの作成とroomsリストへの
      登録
    }
   }
}

blocks.AddRange(rooms.Select(x => x.Value));      //
ラムダ式 roomsに登録されたオブジェクトをすべて
blocksにも登録する

var fixedconditionrooms = new Dictionary<string,
FixedConditionRoomBlock>();
if (data.Tables["条件固定室"] != null)
{
 foreach (DataRow row in data.Tables["条件固定室
 "].Rows)
 {
  string No = Convert.ToString(row[0]);
  if (No != "")
  {
   string Name = Convert.ToString(row[1]);

   IList<ISpaceBlock> referencespaces = new
   List<ISpaceBlock>();
   IList<double> ratios = new List<Double>();

   for (int i = 0; i < 100; i++)
   {
    string spacename = Convert.ToString(row[2 * i
    + 2]);
    string ratio = Convert.ToString(row[2 * i +
    3]);
    if (string.IsNullOrEmpty(ratio) || spacename
    == "END")
     break;
    else
    {
     if (spacename == "Outside" || spacename ==
     "")
     {
      referencespaces.Add(outside);
      ratios.Add(Convert.ToDouble(ratio));
     }

     else if (rooms.ContainsKey(spacename))
     {
      referencespaces.Add(rooms[spacename]);
      ratios.Add(Convert.ToDouble(ratio));
     }
    }

   }
   FixedConditionRoomBlock fixedconditionroom =
   new FixedConditionRoomBlock(Name,
   referencespaces, ratios);
   fixedconditionrooms.Add(Name,
   fixedconditionroom);
  }
 }
}
blocks.AddRange(fixedconditionrooms.Select(x =>
x.Value));


#endregion

#region 発熱体
var heatgenerators = new Dictionary<string,
HeatGenerator>();
if (data.Tables["発熱体"] != null)
{
 foreach (DataRow row in data.Tables["発熱体
 "].Rows)
 {
  string No = Convert.ToString(row[0]);

  if (No != "")
  {
   string name = Convert.ToString(row[1]);
   string sensible = Convert.ToString(row[2]);
   string vapor = Convert.ToString(row[3]);
   //[kg/h]
   string convectionrate =
   Convert.ToString(row[4]);

   string[,] scheduleid = new string[3, 2];
   for (int term = 0; term < 3; term++)
   {
    for (int wday = 0; wday < 2; wday++)
    {
     scheduleid[term, wday] =
     Convert.ToString(row[2 * term + wday + 5]);
     ///row[5]～[10]
    }
   }

   IList<RoomBlock> applyrooms = new
   List<RoomBlock>();

   ///適用室(複数)の取得
   for (int i = 0; i < 100; i++)
   {
    string roomname = Convert.ToString(row[11 +
    i]);
    if (string.IsNullOrEmpty(roomname))
     break;
```

```
      else
      {
       if (rooms.ContainsKey(roomname))
       {
        applyrooms.Add(rooms[roomname]);
       }
      }
     }


     ///スケジュール実体の取得
     ScheduleSeries[,] scseries = new
     ScheduleSeries[3, 2];
     for (int term = 0; term < 3; term++)
     {
      for (int wday = 0; wday < 2; wday++)
      {
       if (schedules.ContainsKey(scheduleid[term,
       wday]))
        scseries[term, wday] =
        schedules[scheduleid[term, wday]];
       else
        scseries[term, wday] = null;
      }
     }


     HeatGenerator heatgen = new HeatGenerator(name,
      Convert.ToDouble(sensible),
      Convert.ToDouble(vapor) / 3600.0,
      Convert.ToDouble(convectionrate),
      scseries, applyrooms);

     heatgenerators.Add(No, heatgen);
    }
   }
}
#endregion

#region エアコン
var aircons = new Dictionary<string,
AirConditionerBlock>();
if (data.Tables["エアコン"] != null)
{
 foreach (DataRow row in data.Tables["エアコン
 "].Rows)
 {
  string No = Convert.ToString(row[0]);
  if (No != "")
  {
   string name = Convert.ToString(row[1]);
   string roomname = Convert.ToString(row[2]);
   string heatingcapacity =
   Convert.ToString(row[3]);
   string coolingcapacity =
   Convert.ToString(row[4]);
   string airvolume = Convert.ToString(row[5]);
   string bypassfactor = Convert.ToString(row[6]);

   string[,] scheduleid = new string[3, 2];
   for (int term = 0; term < 3; term++)
   {
    for (int wday = 0; wday < 2; wday++)
    {
     scheduleid[term, wday] =
     Convert.ToString(row[2 * term + wday + 7]);
     ///row[7]～[12]
    }
   }

   string worktype = Convert.ToString(row[13]);

   RoomBlock room = new RoomBlock();
   if (rooms.ContainsKey(roomname))
    room = rooms[roomname];

   ///スケジュール実体の取得
   ScheduleSeries[,] scseries = new
   ScheduleSeries[3, 2];
   for (int term = 0; term < 3; term++)
   {
    for (int wday = 0; wday < 2; wday++)
    {
     if (schedules.ContainsKey(scheduleid[term,
     wday]))
      scseries[term, wday] =
      schedules[scheduleid[term, wday]];
     else
      scseries[term, wday] = null;
    }
   }

   AirConditionerBlock ac = new
   AirConditionerBlock(
    name, room, outside,
    Convert.ToDouble(heatingcapacity),
    Convert.ToDouble(coolingcapacity),
    Convert.ToDouble(airvolume) / 3600.0,
    Convert.ToDouble(bypassfactor), scseries);

   blocks.Add(ac);
  }
 }

}


#endregion

#region 在室者
var persons = new Dictionary<string,
PersonBlock>();
if (data.Tables["在室者"] != null)
{
 foreach (DataRow row in data.Tables["在室者
 "].Rows)
 {
  string No = Convert.ToString(row[0]);
  //
  if (No != "")
  {
   string total = Convert.ToString(row[1]);
   //一人当たり最大全熱量[W/人]
   string stdsensible = Convert.ToString(row[2]);
   //一人当たり顕熱発熱量[W/人], 24℃の場合
   string slope = Convert.ToString(row[3]);
   //
   string convectionrate =
   Convert.ToString(row[4]);   //対流成分比率

   string[,] scheduleid = new string[3, 2];
   //スケジュールID
   for (int term = 0; term < 3; term++)
   {
```

```
      for (int wday = 0; wday < 2; wday++)
      {
       scheduleid[term, wday] =
       Convert.ToString(row[2 * term + wday + 5]);
       ///row[5]～[10]
      }
     }

     string roomname = Convert.ToString(row[11]);
     RoomBlock room = new RoomBlock();
     if (rooms.ContainsKey(roomname))
      room = rooms[roomname];

     ///スケジュール実体の取得
     ScheduleSeries[,] scseries = new
     ScheduleSeries[3, 2];
     for (int term = 0; term < 3; term++)
     {
      for (int wday = 0; wday < 2; wday++)
      {
       if (schedules.ContainsKey(scheduleid[term,
       wday]))
        scseries[term, wday] =
        schedules[scheduleid[term, wday]];
       else
        scseries[term, wday] = null;
      }
     }

     PersonBlock person = new PersonBlock(
      Convert.ToDouble(total),
      Convert.ToDouble(stdsensible),
      Convert.ToDouble(slope),
      Convert.ToDouble(convectionrate), scseries,
      room);

     blocks.Add(person);
    }
   }
}

#endregion

#region 材料テーブル
var materials = new Dictionary<string, Material>();
//キー(string:材料No)と値(Materialのインスタンス)の
組み合わせ
foreach (DataRow row in data.Tables["材料"].Rows)
{
 string No = Convert.ToString(row[0]);
 // 材料No.
 string Name = Convert.ToString(row[1]);
 // 材料名
 string 熱伝導率 = Convert.ToString(row[2]);
 // [W/mK]
 string 空気層熱抵抗 = Convert.ToString(row[3]);
 // [m2K/W]
 string 容積比率 = Convert.ToString(row[4]);
 // [kJ/m3K]
 string 密度 = Convert.ToString(row[5]);
 // [kg/m3]
 if (!String.IsNullOrWhiteSpace(No)
  && !String.IsNullOrWhiteSpace(Name)
  && !String.IsNullOrWhiteSpace(熱伝導率)
  && !String.IsNullOrWhiteSpace(空気層熱抵抗)
  && !String.IsNullOrWhiteSpace(容積比率)
  && !String.IsNullOrWhiteSpace(密度))
 {
  materials.Add(No,
   new Material
   {
    Name = Name,
    熱伝導率 = Convert.ToDouble(熱伝導率),
    空気層熱抵抗 = Convert.ToDouble(空気層熱抵抗),
    容積比熱 = Convert.ToDouble(容積比率) *
    1000.0,
    密度 = Convert.ToDouble(密度)
   });                                          //材
   料インスタンスの作成とmaterialsへの登録
 }
}
#endregion

#region  壁仕様
var wallSpecs = new Dictionary<string, WallSpec>();
//キー(層構成名)と値(WallSpec)の組み合わせ
string No_old = "";
string No_new = "";
foreach (DataRow row in data.Tables["壁仕様"].Rows)
{
 No_old = No_new;
 No_new = Convert.ToString(row[0]);
 // 壁仕様No.
 string secname = Convert.ToString(row[1]);
 // 断面名・No.
 string name = Convert.ToString(row[2]);
 // 壁仕様名
 string secratio = Convert.ToString(row[3]);
 // 断面比率
 bool newspec = true;
 // 新規仕様かどうか

 if (No_old == No_new) newspec = false;

 if (!String.IsNullOrEmpty(name))
 {
  int count = Convert.ToInt32(row[4]);       //材料
  数

  /// 断面の登録
  var wallsection = new WallSection(secname,
  Convert.ToDouble(secratio), count);
  for (int i = 0; i < count; i++)
  {
   wallsection.materiallist.Add(new
   WallMaterialList
   {
    Material = materials[Convert.ToString(row[5 +
    i * 2])],
    Thickness = Convert.ToDouble(row[6 + i * 2]) /
    1000.0
   });
  }


  /// 層構成の新規生成
  if (newspec == true)
  {
   wallSpecs.Add(name,
    new WallSpec
    {
     Name = name,
     SectionNum = 0
    });
```

```
    }

    WallSpec wallspec = wallSpecs[name];
    wallspec.SectionNum++;
    wallspec.wallsection.Add(wallsection);

  }
}
#endregion

#region 窓仕様
var windowProperties = new Dictionary<string,
WindowProperty>();
foreach (DataRow row in data.Tables["窓仕様"].Rows)
{
  string No = Convert.ToString(row[0]);
  string 熱貫流率 = Convert.ToString(row[1]);
  string ηR = Convert.ToString(row[2]);
  string ηC = Convert.ToString(row[3]);
  string 厚さ = Convert.ToString(row[4]);

  if (!String.IsNullOrWhiteSpace(No)
   && !String.IsNullOrWhiteSpace(熱貫流率)
   && !String.IsNullOrWhiteSpace(ηR)
   && !String.IsNullOrWhiteSpace(ηC)
   && !String.IsNullOrWhiteSpace(厚さ))
  {
    windowProperties.Add(No,
      new WindowProperty
      {
        UValue = Convert.ToDouble(熱貫流率),
        EtaR = Convert.ToDouble(ηR),
        EtaC = Convert.ToDouble(ηC),
        Thickness = Convert.ToDouble(厚さ) / 1000.0
      });

  }
}


#endregion

#region 壁ブロック
///複数断面の壁は,1断面ごとにインスタンスを作成する
var walls = new Dictionary<string, IBlock>();    //
キー(string:No.)と値(IBlock:WallBlockのインスタンス)
の組み合わせ
foreach (DataRow row in data.Tables["壁"].Rows)
{
  string No = Convert.ToString(row[0]);
  if (No != "")
  {
  string 居室A = Convert.ToString(row[1]);
  string 居室B = Convert.ToString(row[2]);        //
  空白の場合は外部環境
  double 方位角 = Convert.ToDouble(row[3]);
  double 傾斜角 = Convert.ToDouble(row[4]);
  string 面積 = Convert.ToString(row[5]);
  string 床フラグ = Convert.ToString(row[6]);
  string 壁仕様名 = Convert.ToString(row[7]);
  string 壁仕様向き = Convert.ToString(row[8]);
  string A側αc = Convert.ToString(row[9]);
  string A側αr = Convert.ToString(row[10]);
  string A側日射吸収率 = Convert.ToString(row[11]);
  string A側長波放射率 = Convert.ToString(row[12]);
  string B側αc = Convert.ToString(row[13]);
  string B側αr = Convert.ToString(row[14]);
  string B側日射吸収率 = Convert.ToString(row[15]);
  string B側長波放射率 = Convert.ToString(row[16]);


  if (!String.IsNullOrWhiteSpace(No)
   && !String.IsNullOrWhiteSpace(居室A)
   && !String.IsNullOrWhiteSpace(面積)
   && !String.IsNullOrWhiteSpace(壁仕様名)
   && !String.IsNullOrWhiteSpace(壁仕様向き))
  {

   var roomA = rooms[居室A] as RoomBlock;

   ISpaceBlock roomB = null;
   if (居室B == "")
   {
    居室B = "外気";
    roomB = outside;
   }
   else if (rooms.ContainsKey(居室B))
   {
    roomB = rooms[居室B] as RoomBlock;
   }

   else if (fixedconditionrooms.ContainsKey(居室B))
   {
    roomB = fixedconditionrooms[居室B] as
    FixedConditionRoomBlock;
   }

   ///断面ごとにブロック作成
   WallSpec wallspec = new WallSpec();
   wallspec = wallSpecs[壁仕様名];
   int secnum = wallspec.SectionNum;
   double totalratio = 0.0;
   for (int i = 0; i < secnum; i++)
    totalratio +=
    wallspec.wallsection[i].SectionAreaRatio;

    for (int i = 0; i < secnum; i++)
    {
     WallSection wallsection = new WallSection();
     wallsection = wallspec.wallsection[i];
     double ratio = wallsection.SectionAreaRatio;
     string secname = wallsection.SecName;

     //壁
     var wall = new WallBlock(
      name: String.Format("{0}-{1}-{2}-{3}", No,
      居室A, 居室B, 壁仕様名),
      spaceA: roomA,
      spaceB: roomB,
      area: Convert.ToDouble(面積) * ratio /
      totalratio,
      floorofspaceA: Convert.ToInt32(床フラグ),
      materialList:
      wallsection.materiallist.Select(x =>
      x.Material).ToArray(),
      thicknessList:
      wallsection.materiallist.Select(x =>
      x.Thickness).ToArray(),
      matdirection: Convert.ToInt32(壁仕様向き),
      alphacA: Convert.ToDouble(A側αc),
      alpharA: Convert.ToDouble(A側αr),
      solarabsorptionrateA: Convert.ToDouble(A側日
      射吸収率),
      longwaveemissionrateA: Convert.ToDouble(A側
```

```
            長波放射率),
            alphacB: Convert.ToDouble(B側αc),
            alpharB: Convert.ToDouble(B側αr),
            solarabsorptionrateB: Convert.ToDouble(B側日
            射吸収率),
            longwaveemissionrateB: Convert.ToDouble(B側
            長波放射率),
            faceh: Convert.ToDouble(傾斜角),
            facea: Convert.ToDouble(方位角)
          );
          //WallBlockインスタンスの生成
          walls.Add(No + secname, wall);
          //wallsリストへの登録

          //A側
          blocks.Add(wall.FaceA);

          if (roomA is FixedConditionRoomBlock)
          {
            blocks.Add(new
            TotalHeatTransferBlock(wall.FaceA));
          }
          else
          {
            blocks.Add(new
            ConvectionHeatTransferBlock(wall.FaceA));
            blocks.Add(new
            ConvectionMoistureTransferBlock(wall.FaceA))
            ;
          }

          blocks.Add(wall.FaceB);
          if (roomB is OutsideBlock)           //B側が外
          部
          {
            blocks.Add(new
            TotalHeatTransferBlock(wall.FaceB));
          }
          else if (roomB is FixedConditionRoomBlock)
          {
            blocks.Add(new
            TotalHeatTransferBlock(wall.FaceA));
          }
          else
          {
            blocks.Add(new
            ConvectionHeatTransferBlock(wall.FaceB));
            blocks.Add(new
            ConvectionMoistureTransferBlock(wall.FaceB))
            ;
          }
        }
      }
    }
  }
}
blocks.AddRange(walls.Select(x => x.Value));
//wallsに登録されたオブジェクトをすべてblocksにも登
録する
#endregion

#region 窓ブロック
var windows = new Dictionary<string, IBlock>();
// キー(string:No.)と値(IBlock:WindowBlockのインスタ
ンス)の組み合わせ
if (data.Tables["窓"] != null)
{
  foreach (DataRow row in data.Tables["窓"].Rows)
  {
    string No = Convert.ToString(row[0]);
    if (No != "")
    {
      string 居室A = Convert.ToString(row[1]);
      string 居室B = Convert.ToString(row[2]);
      string 方位角 = Convert.ToString(row[3]);
      string 傾斜角 = Convert.ToString(row[4]);
      string 縦寸法 = Convert.ToString(row[5]);
      ///[m]
      string 横寸法 = Convert.ToString(row[6]);
      ///[m]
      string 窓仕様No = Convert.ToString(row[7]);
      ///Noで指定


      if (!String.IsNullOrWhiteSpace(No)
        && !String.IsNullOrWhiteSpace(居室A)
        && !String.IsNullOrWhiteSpace(縦寸法)
        && !String.IsNullOrWhiteSpace(横寸法)
        && !String.IsNullOrWhiteSpace(窓仕様No))
      {
        //居室A,B
        var roomA = rooms[居室A] as RoomBlock;
        ISpaceBlock roomB;
        if (居室B == "")
        {
          居室B = "外気";
          roomB = outside;
        }
        else
        {
          roomB = rooms[居室B] as RoomBlock;
        }

        //窓
        var window = new WindowBlock(
          name: String.Format("Win{0}-{1}-{2}", No, 居
          室A, 居室B),
          spaceA: roomA,
          spaceB: roomB,
          width: Convert.ToDouble(縦寸法),
          height: Convert.ToDouble(横寸法),
          uvalue: windowProperties[窓仕様No].UValue,
          etar: windowProperties[窓仕様No].EtaR,
          etac: windowProperties[窓仕様No].EtaC,
          thickness: windowProperties[窓仕様
          No].Thickness,
          faceh: Convert.ToDouble(傾斜角),
          facea: Convert.ToDouble(方位角)
        );
        windows.Add(No, window);

        //A側は必ず居室
        blocks.Add(window.FaceA);
        blocks.Add(new
        ConvectionHeatTransferBlock(window.FaceA));
        blocks.Add(new
        ConvectionMoistureTransferBlock(window.FaceA))
        ;

        blocks.Add(window.FaceB);
        if (roomB is OutsideBlock)           //B側が外
        部
        {
          blocks.Add(new
          TotalHeatTransferBlock(window.FaceB));
```

```
      }
      else
      {
       blocks.Add(new
       ConvectionHeatTransferBlock(window.FaceB));
       blocks.Add(new
       ConvectionMoistureTransferBlock(window.FaceB
       ));
      }
     }
    }
   }
   blocks.AddRange(windows.Select(x => x.Value));
}
#endregion

#region ドアブロック
var doors = new Dictionary<string, IBlock>();
if (data.Tables["ドア"] != null)
{
 foreach (DataRow row in data.Tables["ドア"].Rows)
 {
  string No = Convert.ToString(row[0]);
  if (No != "")
  {
   string 居室A = Convert.ToString(row[1]);
   string 居室B = Convert.ToString(row[2]);
   string 方位角 = Convert.ToString(row[3]);
   string 傾斜角 = Convert.ToString(row[4]);
   string 縦寸法 = Convert.ToString(row[5]);
   ///[m]
   string 横寸法 = Convert.ToString(row[6]);
   ///[m]
   string 熱貫流率 = Convert.ToString(row[7]);
   ///
   string 材料No = Convert.ToString(row[8]);
   string 材料厚さ = Convert.ToString(row[9]);
   ///[mm]

   if (!String.IsNullOrWhiteSpace(No)
    && !String.IsNullOrWhiteSpace(居室A)
    && !String.IsNullOrWhiteSpace(縦寸法)
    && !String.IsNullOrWhiteSpace(横寸法))
   {
    //居室A,B
    var roomA = rooms[居室A] as RoomBlock;
    ISpaceBlock roomB;
    if (居室B == "")
    {
     居室B = "外気";
     roomB = outside;
    }
    else
    {
     roomB = rooms[居室B] as RoomBlock;
    }

    Material material = materials[材料No];

    var door = new DoorBlock(
     name: String.Format("Dor{0}-{1}-{2}", No, 居
     室A, 居室B),
     spaceA: roomA,
     spaceB: roomB,
     width: Convert.ToDouble(縦寸法),
     height: Convert.ToDouble(横寸法),
     uvalue: Convert.ToDouble(熱貫流率),
     faceh: Convert.ToDouble(傾斜角),
     facea: Convert.ToDouble(方位角),
     material: material,
     thickness: Convert.ToDouble(材料厚さ) /
     1000.0
     );
    doors.Add(No, door);

    //A側は必ず居室
    blocks.Add(door.FaceA);
    blocks.Add(new
    ConvectionHeatTransferBlock(door.FaceA));
    blocks.Add(new
    ConvectionMoistureTransferBlock(door.FaceA));

    blocks.Add(door.FaceB);

    if (roomB is OutsideBlock)          //B側が外
    部
    {
     blocks.Add(new
     TotalHeatTransferBlock(door.FaceB));
    }
    else
    {
     blocks.Add(new
     ConvectionHeatTransferBlock(door.FaceB));
     blocks.Add(new
     ConvectionMoistureTransferBlock(door.FaceB))
     ;
    }
   }
  }
 }
 blocks.AddRange(doors.Select(x => x.Value));
}


#endregion


#region 換気ブロック
var ventilations = new Dictionary<String,
IBlock>();
if (data.Tables["換気"] != null)
{
 foreach (DataRow row in data.Tables["換気"].Rows)
 {
  string No = Convert.ToString(row[0]);
  if (No != "")
  {
   string 居室A = Convert.ToString(row[1]);
   //空白の場合は外部環境
   string 居室B = Convert.ToString(row[2]);
   //空白の場合は外部環境
   string 風量 = Convert.ToString(row[3]);

   string[,] scheduleid = new string[3, 2];
   for (int term = 0; term < 3; term++)
   {
    for (int wday = 0; wday < 2; wday++)
    {
     scheduleid[term, wday] =
     Convert.ToString(row[2 * term + wday + 4]);
     ///row[4]～[9]
    }
   }
```

```
      ISpaceBlock roomA = null;
      ISpaceBlock roomB = null;
      if (居室A == "" || 居室A == "Outside")
      {
       居室A = "外気";
       roomA = outside;
      }
      else if (fixedconditionrooms.ContainsKey(居室
      A))
      {
       roomA = fixedconditionrooms[居室A] as
       FixedConditionRoomBlock;
      }
      else
      {
       roomA = rooms[居室A] as RoomBlock;
      }

      if (居室B == "" || 居室B == "Outside")
      {
       居室B = "外気";
       roomB = outside;
      }
      else if (fixedconditionrooms.ContainsKey(居室
      B))
      {
       roomA = fixedconditionrooms[居室B] as
       FixedConditionRoomBlock;
      }
      else
      {
       roomB = rooms[居室B] as RoomBlock;
      }

      ///スケジュール実体の取得
      ScheduleSeries[,] scseries = new
      ScheduleSeries[3, 2];
      for (int term = 0; term < 3; term++)
      {
       for (int wday = 0; wday < 2; wday++)
       {
        if (schedules.ContainsKey(scheduleid[term,
        wday]))
         scseries[term, wday] =
         schedules[scheduleid[term, wday]];
        else
         scseries[term, wday] = null;
       }
      }

      var ventilation = new
      VentilationHeatTransferBlock(
       upperspace: roomA,
       lowerspace: roomB,
       airvolume: Convert.ToDouble(風量) / 3600.0,
       schedule: scseries
       );
      ventilations.Add(No, ventilation);
     }
    }
    blocks.AddRange(ventilations.Select(x =>
    x.Value));
  }
  #endregion
```

```
      excelReader.Close();

      return blocks;
     }

    }
   }
  }
```

1.2 Simulator.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace HLFX
{
 using Block;

 public class Simulator
 {

  /// <summary>
  /// 計算時間間隔[s]
  /// 原則60s
  /// 壁,窓などはさらに細かい時間間隔で行う
  /// </summary>
  public long Dt { get; set; }

  /// <summary>
  /// 開始日時
  /// </summary>
  public DateTime StartDay { get; set; }

  /// <summary>
  /// 終了日時
  /// </summary>
  public DateTime StopDay { get; set; }

  public IList<IBlock> Blocks { get; set; }

  /// <summary>
  /// ロガー
  /// </summary>
  public Action<SimulationContext> Logger { get; set; }

  public Simulator()
  {
   Dt = 60 ;             ///初期値60s

   StartDay = new DateTime(2015, 7, 5, 0, 0, 0);
   StopDay = new DateTime(2015, 7, 31, 23, 50, 0);
  }

  public void Run()
  {
   long elapsed = 0L;
   long totalClock = GetTotalLoopCount();

   var context = new SimulationContext()
   {
```

```
  Dt = Dt,
  Elasped = 0L,
  Current = StartDay,
  Before = StartDay,
  Blocks = Blocks,
};

DateTime heatingtermstartday = new DateTime(2014, 11,
30);
DateTime heatingtermendday = new DateTime(2015, 1,
2);
DateTime coolingtermstartday = new DateTime(2015, 6,
1);
DateTime coolingtermendday = new DateTime(2015, 9,
30);
context.SetTerm(heatingtermstartday,
heatingtermendday, coolingtermstartday,
coolingtermendday);

InitBlocks(context);

for (long i = 0; i < totalClock; i++, elapsed += Dt)
{
  context.Before = context.Current;
  context.Current = StartDay.AddSeconds(elapsed);
  context.Elasped = elapsed;

  ///
  ///気象データ更新
  ///外部条件は必ず最初のブロックとしておく
  ///
  Blocks[0].IterationInit(context);

  ///
  ///表面への日射セット
  ///
  foreach (var kernel in Blocks)
  {
    if (kernel is WallFaceConnector)
   kernel.IterationInit(context);
  }

  ///
  ///窓の透過日射計算
  ///
  foreach (var kernel in Blocks)
  {
    if (kernel is WindowBlock)
      kernel.IterationInit(context);
  }

  ///
  ///室の内部発熱・除去熱量計算(エアコン判定含む)
  ///
  foreach (var kernel in Blocks)
  {
    if (kernel is RoomBlock)
      kernel.IterationInit(context);
  }

  ///
  ///条件固定室の温度計算
  ///
  foreach (var kernel in Blocks)
  {
    if (kernel is FixedConditionRoomBlock)
      kernel.IterationInit(context);
  }


  ///
  ///エアコン処理熱量熱量計算
  ///
  foreach (var kernel in Blocks)
  {
    if (kernel is AirConditionerBlock)
      kernel.Run(context);
  }

  ///
  ///対流・総合熱伝達
  ///
  foreach (var kernel in Blocks)
  {
    if (kernel is ConvectionHeatTransferBlock ||
   kernel is TotalHeatTransferBlock)
      ///
      ///WallFaceのQcに値をセットする
      ///
      kernel.Run(context);
  }


  ///
  ///換気
  ///
  foreach (var kernel in Blocks)
  {
    if (kernel is VentilationHeatTransferBlock)
      kernel.Run(context);
  }

  ///
  ///窓の伝熱計算
  ///透過日射は計算済み
  ///
  foreach (var kernel in Blocks)
  {
    if (kernel is WindowBlock)
      kernel.Run(context);
  }

  ///
  ///ドアの伝熱計算
  ///
  foreach (var kernel in Blocks)
  {
    if (kernel is DoorBlock)
      kernel.Run(context);
  }

  ///
  ///壁の計算
  ///
  foreach (var kernel in Blocks)
  {
    if (kernel is WallBlock)
      kernel.Run(context);
  }

  ///
  ///居室の計算
  ///
  foreach (var kernel in Blocks)
```

```
      {
       if (kernel is RoomBlock)
        kernel.Run(context);
      }

      if (Logger != null)
      {
       Logger(context);
      }

      foreach (var obj in Blocks)
      {
       obj.Commit(context);
      }

    }
   }

   private void InitBlocks(SimulationContext context)
   {
    foreach (var block in context.Blocks)
    {

     block.Init(context);

    }
    int n = 0;

   }

   public long GetTotalLoopCount()
   {
    return (long)Math.Ceiling((StopDay -
    StartDay).TotalSeconds / Dt);
   }
  }
}
```

1.3 SimulationContext.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using HLFX.Block;

namespace HLFX
{
 public class SimulationContext
 {

  /// <summary>
  /// 計算時間間隔[s]
  /// 原則60s
  /// 壁,窓などはさらに細かい時間間隔で行う
  /// </summary>
  public long Dt { get; set; }

  /// <summary>
  /// 現日付・時刻
  /// </summary>
  public DateTime Current { get; internal set; }

  /// <summary>
  /// 前時点日付・時刻
  /// </summary>
  public DateTime Before { get; internal set; }

  /// <summary>
  /// 現時刻(時刻・分を4桁の整数化したもの)
  /// </summary>
  public int CurrentTime
  {
   get
   {
    return Current.Hour * 100 + Current.Minute;
   }
  }
  /// <summary>
  /// 前時点の時刻(時刻・分を4桁の整数化したもの)
  /// </summary>
  public int BeforeTime
  {
   get
   {
    return Before.Hour* 100 + Before.Minute;
   }
  }

  /// <summary>
  /// 期の定義 0:暖房期, 1:冷房期, 2:中間期
  /// </summary>
  public int[] Term { get; set; }

  /// <summary>
  /// 曜日の定義 0:平日, 1:休日
  /// </summary>
  public int[] DayoftheWeek { get; set; }

  /// <summary>
  /// 現在の期を返す
  /// 0:暖房期, 1:冷房期, 2:中間期
  /// </summary>
  public int CurrentTerm
  {
   get
   {
    DateTime date = new DateTime(2015, 1, 1);
    TimeSpan ts = Current.Date - date;
    return Term[ts.Days];
   }
  }
  /// <summary>
  /// 前時点の期を返す
  /// 0:暖房期, 1:冷房期, 2:中間期
  /// </summary>
  public int BeforeTerm
  {
   get
   {
    DateTime date = new DateTime(2015, 1, 1);
    TimeSpan ts = Before.Date - date;
    return Term[ts.Days];
   }
  }


  /// <summary>
  /// 経過時間[秒]
```

```
/// </summary>
public long Elasped { get; internal set; }
public IList<IBlock> Blocks { get; internal set; }

/// <summary>
/// 現在の曜日を返す
/// 0(日), 6(土), 7(祝)は1を,それ以外は0を返す
/// </summary>
public int CurrentWeek
{
 get
 {
  DateTime date = new DateTime(2015, 1, 1);
  TimeSpan ts = Current.Date - date;
  if (1 <= DayoftheWeek[ts.Days] &&
  DayoftheWeek[ts.Days] <= 5)
   return 0;
  else
   return 1;
 }
}

/// <summary>
/// 前時点の曜日を返す
/// 0(日), 6(土), 7(祝)は1を,それ以外は0を返す
/// </summary>
public int BeforeWeek
{
 get
 {
  DateTime date = new DateTime(2015, 1, 1);
  TimeSpan ts = Before.Date - date;
  if (1 <= DayoftheWeek[ts.Days] &&
  DayoftheWeek[ts.Days] <= 5)
   return 0;
  else
   return 1;
 }
}

public void SetTerm(DateTime heatingtermstartday,
DateTime heatingtermendday, DateTime
coolingtermstartday, DateTime coolingtermendday)
{
 Term = new int[365];
 DayoftheWeek = new int[365];

 for (int i = 0; i < 365; i++)
 {
  Term[i] = 2;           ///中間期で初期化
  DayoftheWeek[i] = 0;   ///平日で初期化
 }

 DateTime sd = new DateTime(2015, 1, 1);       ///1月1
 日
 DateTime ed = new DateTime(2014, 12, 31);     ///(前
 年)12月31日

 ///
 ///暖房期の設定
 ///
 if (heatingtermstartday < ed)                 ///暖房
 期間が年をまたぐ場合
 {
  TimeSpan a = heatingtermendday - sd;     ///1月1日
  から暖房終了日まで
  for (int i = 0; i <= a.Days; i++)
   Term[i] = 0;

  TimeSpan b = ed - heatingtermstartday;   ///暖房開
  始日(前年)から12月31日まで
  for (int i = 0; i <= b.Days; i++)
   Term[364-i] = 0;
 }
 else
 {

  TimeSpan a = heatingtermstartday - sd;
  TimeSpan b = heatingtermendday - sd;

  for (int i = a.Days; i <= b.Days; i++)
   Term[i] = 0;
 }

 ///
 ///冷房期の設定
 ///
 if (coolingtermstartday < ed)                 ///冷房
 期間が年をまたぐ場合
 {
  TimeSpan a = coolingtermendday - sd;     ///1月1日
  から冷房終了日まで
  for (int i = 0; i <= a.Days; i++)
   Term[i] = 1;

  TimeSpan b = ed - coolingtermstartday;   ///暖房開
  始日(前年)から12月31日まで
  for (int i = 0; i <= b.Days; i++)
   Term[364 - i] = 1;
 }
 else
 {

  TimeSpan a = coolingtermstartday - sd;
  TimeSpan b = coolingtermendday - sd;

  for (int i = a.Days; i <= b.Days; i++)
   Term[i] = 1;
 }

 ///
 ///曜日の設定
 ///1月1日を日曜とする
 ///
 int ii = 0;
 for (int i = 0; i < 365; i++)
 {
  DayoftheWeek[i] = ii++;     //代入後にiiをインクリ
  メント
  if (ii == 7) ii = 0;        //7になったら0に戻す
 }

 ///祝日
 List<DateTime> DateList = new List<DateTime>();

 DateList.Add(new DateTime(2015, 1, 1));
 DateList.Add(new DateTime(2015, 1, 15));
 DateList.Add(new DateTime(2015, 2, 11));
 DateList.Add(new DateTime(2015, 3, 21));
 DateList.Add(new DateTime(2015, 4, 29));
 DateList.Add(new DateTime(2015, 5, 3));
 DateList.Add(new DateTime(2015, 5, 4));
 DateList.Add(new DateTime(2015, 5, 5));
 DateList.Add(new DateTime(2015, 7, 20));
```

```
    DateList.Add(new DateTime(2015, 9, 15));
    DateList.Add(new DateTime(2015, 9, 23));
    DateList.Add(new DateTime(2015, 10, 10));
    DateList.Add(new DateTime(2015, 11, 3));
    DateList.Add(new DateTime(2015, 11, 23));
    DateList.Add(new DateTime(2015, 12, 23));

    for (int n = 0; n < DateList.Count; n++)
    {
     TimeSpan a = DateList[n] - sd;
     DayoftheWeek[a.Days] = 7;
    }

   }

   /// <summary>
   /// 日付を1/1からの通算日に変換する
   /// </summary>
   public static int GetTotalDay(DateTime today)
   {
    DateTime nd = new DateTime(2015, 1, 1);
    TimeSpan a = today - nd;
    return a.Days + 1;
   }

  }
}
```

1.4 IBlock.cs

```
namespace HLFX.Block
{
 public interface IBlock
 {
  bool IsComposite { get; }
  string Name { get; set; }
  IConnector[] GetConnectors();

  /// <summary>
  /// オブジェクトごとの計算時間間隔 単位注意[ms]
  /// </summary>
  /// <param name="context"></param>
  long Dt { get; set; }

  /// <summary>
  /// 初期化 (一度だけ)
  /// </summary>
  /// <param name="context"></param>
  void Init(SimulationContext context);

  /// <summary>
  /// 初期化 (反復前に毎回))
  /// </summary>
  /// <param name="context"></param>
  void IterationInit(SimulationContext context);

  /// <summary>
  /// 計算実行
  /// </summary>
  /// <param name="context"></param>
  void Run(SimulationContext context);

  /// <summary>
  /// 後処理 (反復後に毎回))
  /// </summary>
  /// <param name="context"></param>
  void Commit(SimulationContext context);


 }
}
```

1.5 IConnector.cs

```
namespace HLFX.Block
{
 public interface IConnector
 {
  string Name { get; set; }

  void AddHeat(double Qc);
  void AddMoisture(double dJ, IConnector src, IConnector
  kernel);
  void AddMoisture(double dJ);
  double GetHeat();
  double GetMoisture();
  void Run(SimulationContext context);
  void Commit(SimulationContext context);
 }
}
```

1.6 BaseBlock.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace HLFX.Block
{
 public abstract class BaseBlock : IBlock, IConnector
 {
  public string Name { get; set; }

  /// <summary>
  /// オブジェクトごとの計算時間間隔  単位[ms]
  /// </summary>
  public virtual long Dt { get; set; }

  public virtual bool IsComposite { get { return
  false; } }

  public virtual IConnector[] GetConnectors()
  {
   return null;
  }

  /// <summary>
  /// 内部熱量[J]
  /// </summary>
  protected double U;

  /// <summary>
```

```
/// 熱移動量[J]
/// </summary>
protected double dU_tick;

/// <summary>
/// 内部水分量[kg]
/// </summary>
protected double J;

/// <summary>
/// 水分移動量[kg]
/// </summary>
protected double dJ_tick;

/// <summary>
/// 熱移動
/// </summary>
/// <param name="dU">熱移動量[J]</param>
public virtual void AddHeat(double dU, IConnector src,
IConnector kernel)
{

public virtual void AddHeat(double dU)
{
 dU_tick += dU;
}

public virtual double GetHeat()
{
 return dU_tick;
}

/// <summary>
/// 水分移動
/// </summary>
/// <param name="dJ">水分伝達量（単位[kg]）</param>
public virtual void AddMoisture(double dJ, IConnector
src, IConnector kernel)
{
 dJ_tick += dJ;
 System.Diagnostics.Debug.Assert(dJ != -0.0002);
}

public virtual void AddMoisture(double dJ)
{
 dJ_tick += dJ;
}

public virtual double GetMoisture()
{
 return dJ_tick;
}

public virtual void Init(SimulationContext context)
{
}

public virtual void IterationInit(SimulationContext
context)
{
}

public virtual void Run(SimulationContext context)
{
}

public virtual void Commit(SimulationContext context)
{
 var children = GetConnectors();
 if (children != null)
 {
  foreach (var child in children)
  {
   child.Commit(context);
  }
 }

 U += dU_tick;
 J += dJ_tick;

 dU_tick = 0.0;
 dJ_tick = 0.0;
}

/// <summary>
/// ブロックごとのDtを求める(60sの約数,単位ms)
/// </summary>
/// <param name="dt">ブロックごとの収束条件を満たす
dt(単位ms) </param>
public long CalcDt(long dt)
{
 long[] tmpdt = new long[5];
 tmpdt[0] = 60000;  //60s  × 1回
 tmpdt[1] = 30000;  //30s  × 2回
 tmpdt[2] = 20000;  //20s  × 3回
 tmpdt[3] = 15000;  //15s  × 4回
 tmpdt[4] = 10000;  //10s  × 6回

 //これ以降は[0]～[4]を10および100で除した値とする

 if (dt >= tmpdt[4])
 {
  for (int n = 0; n < 5; n++)
  {
   if (dt >= tmpdt[n])
    return tmpdt[n];
  }
 }
 else if (dt >= tmpdt[4] / 10)
 {
  for (int n = 0; n < 5; n++)
  {
   if (dt >= (tmpdt[n] / 10))
    return tmpdt[n] / 10;
  }
 }
 else if (dt >= tmpdt[4] / 100)
 {
  for (int n = 0; n < 5; n++)
  {
   if (dt >= (tmpdt[n] / 100))
    return tmpdt[n] / 100;
  }
 }
 else if (dt >= tmpdt[4] / 1000)
 {
  for (int n = 0; n < 4; n++)
  {
   if (dt >= (tmpdt[n] / 1000))
    return tmpdt[n] / 1000;
  }
 }

 return 0;
```

```
    }

  }
}
```

1.7 ISpaceBlock.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace HLFX.Block
{
  /// <summary>
  /// 空間
  /// </summary>
  public interface ISpaceBlock : IConnector
  {
    /// <summary>
    /// 温度[℃]
    /// </summary>
    double Temp { get; }

    /// <summary>
    /// MRT[℃]
    /// </summary>
    double MRT { get; set; }

    /// <summary>
    /// 絶対湿度[kg/kg']
    /// </summary>
    double Humidity { get; }

    /// <summary>
    /// 窓から取得する直達日射(放射成分)[W]
    /// 単位注意(窓面積を乗じた和)
    /// </summary>
    double DirectSolarRadiationTotal { get; set; }

    /// <summary>
    /// 窓から取得する直達日射(対流成分)[W]
    /// 単位注意(窓面積を乗じた和)
    /// </summary>
    double DirectSolarConvectionTotal { get; set; }

    double SkySolarRadiationTotal { get; set; }
    double SkySolarConvectionTotal { get; set; }
    double ReflectSolarRadiationTotal { get; set; }
    double ReflectSolarConvectionTotal { get; set; }

    /// <summary>
    /// 壁などの面積当たりに配分される内部発熱(放射成分)を返す [W/m2]
    /// </summary>
    double SensibleHeatGenerationRadiationToWall { get; }

    /// <summary>
    /// 床面積当たりに配分される日射量(放射成分)を返す [W/m2]
    /// </summary>
    double SolarRadiationToFloor { get; }

    /// <summary>
    /// 壁など(床以外)の床面積当たりに配分される日射量(放射成分)を返す [W/m2]
    /// </summary>
    double SolarRadiationToWall { get; }


    /// <summary>
    /// 面の角度を受け取り,当該面に入射する日射量,夜間放射量を返す
    /// </summary>
    void GetSolar(double costheta, double facecosh, out double directsolar, out double skysolar, out double reflectsolar, out double longwave);


    /// <summary>
    /// 面と太陽方向のなす角のcosを返す
    /// </summary>
    double GetCosTheta(double facewz, double faceww, double facews);

    /// <summary>
    /// 壁のSAT温度(日射吸収あり)
    /// </summary>;
    double GetSAT(double solarabsorptionrate, double longwaveemissionrate, double alphat, double costheta, double facecosh);

    /// <summary>
    /// 窓のSAT温度(日射吸収なし)
    /// </summary>
    double GetSAT(double longwaveemissionrate, double alphat, double costheta, double facecosh);

    void SetFaceConnector(WallFaceConnector connectingface, int facetype);

    void SetVentilation(VentilationHeatTransferBlock ventilation);

    void AddVent(double qv, double jv);
  }
}
```

1.8 ILayerBlock.cs

```
using System;
namespace HLFX.Block
{
  public interface ILayerBlock : IConnector
  {
    /// <summary>
    /// 隣接するオブジェクト(空間または外部)
    /// </summary>
    ISpaceBlock Space { get; }

    /// <summary>
    /// 隣接するオブジェクト表面(壁または窓の表面)
    /// </summary>
    ISurface Surface { get; }
```

```
    /// <summary>
    /// 表面積[m^2]
    /// </summary>
    double Area { get; }

    /// <summary>
    /// 対流熱伝達率[W/m2K]
    /// </summary>
    double AlphaC { get; }

    /// <summary>
    /// 放射熱伝達率[W/m2K]
    /// </summary>
    double AlphaR { get; }

    /// <summary>
    /// 日射吸収率[-]
    /// </summary>
    double SolarAbsorptionRate { get; }

    /// <summary>
    /// 長波長放射率[-]
    /// </summary>
    double LongWaveEmissionRate { get; }

    /// <summary>
    /// 水分伝達率[kg/m2s(kg/kg')]
    /// </summary>
    double MoistureTransferRate { get; }

    /// <summary>
    /// 空気温度[℃]
    /// </summary>
    double SpaceTemp { get; }

    /// <summary>
    /// 空気絶対湿度 [kg/kg']
    /// </summary>
    double SpaceHumidity { get; }

    /// <summary>
    /// 壁窓表面温度[℃]
    /// </summary>
    double SurfaceTemp { get; }

    /// <summary>
    /// 壁窓表面絶対湿度 [kg/kg']
    /// </summary>
    double SurfaceHumidity { get; }
  }

  public interface ISurface
  {
    /// <summary>
    /// 壁・窓ブロックのA側表面温度
    /// </summary>
    double TempA { get; }

    /// <summary>
    /// 壁・窓ブロックのB側表面温度
    /// </summary>
    double TempB { get; }

    /// <summary>
    /// 壁・窓ブロックのA側表面絶対湿度
    /// </summary>
    double HumidityA { get; }

    /// <summary>
    /// 壁・窓ブロックのB側表面温度
    /// </summary>
    double HumidityB { get; }

    /// <summary>
    /// A側の熱流を与える(空気→表面の向きを正)
    /// </summary>
    void AddHeatA();

    /// <summary>
    /// B側の熱流を与える(空気→表面の向きを正)
    /// </summary>
    void AddHeatB();

    /// <summary>
    /// A側の水分流を与える(空気→表面の向きを正)
    /// </summary>
    void AddMoistureA();

    /// <summary>
    /// B側の水分流を与える(空気→表面の向きを正)
    /// </summary>
    void AddMoistureB();

    double Costheta { get; set; }
  }
}
```

1.9 RoomBlock.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace HLFX.Block
{

  public class RoomBlock : BaseBlock, ISpaceBlock
  {
    private const double rho_air = 1.2;
    private const double c_air = 1005.0;

    public struct ConnectingFace
    {
      public int FaceType;                ///0:床以外, 1:床
      public WallFaceConnector Face;

      public ConnectingFace(WallFaceConnector face, int
      facetype)
      {
        Face = face;
        FaceType = facetype;
      }
    }


    /// <summary>
    /// 接続する表面の数
```

```
/// </summary>
public int ConnectingFaceNum { get; set; }

/// <summary>
/// 接続する表面
/// </summary>
private List<ConnectingFace> ListConnectingFace = new
List<ConnectingFace>();

/// <summary>
/// 接続する換気経路(自分が下流のものに限る)
/// </summary>
private List<VentilationHeatTransferBlock>
ListVentilation = new
List<VentilationHeatTransferBlock>();

/// <summary>
/// 外部環境への参照
/// </summary>
public OutsideBlock Outside;


/// <summary>
/// 設置する発熱体
/// </summary>
public List<HeatGenerator> ListHeatGenerator = new
List<HeatGenerator>();

/// <summary>
/// 在室者(複数人)
/// </summary>
public PersonBlock Person;

/// <summary>
/// 設置するエアコン　当面1台だけとする
/// </summary>
public AirConditionerBlock AC;

/// <summary>
/// 気積[m3]
/// </summary>
public double Volume { get; set; }

/// <summary>
/// 換気回数(窓CLOSE)
/// </summary>
public double VentNumClose { get; set; }

/// <summary>
/// 換気回数(窓OPEN)
/// </summary>
public double VentNumOpen { get; set; }

/// <summary>
/// 顕熱容量[J/K]
/// </summary>
public double Cs { get; set; }

/// <summary>
/// 潜熱容量[kg/(kg/kg')]
/// </summary>
public double Cl { get; set; }

/// <summary>
/// 表面積合計[m2]
/// </summary>
public double TotalArea { get; set; }

/// <summary>
/// 床として定義された面の面積合計[m2]
/// </summary>
public double TotalFloorArea { get; set; }

/// <summary>
/// 在室スケジュール
/// </summary>
public ScheduleSeries[,] OccupySchedule;


#region 状態

/// <summary>
/// 室空気温度[℃]
/// </summary>
public double Temp { get; set; }

/// <summary>
/// 室絶対湿度 [kg/kg']
/// </summary>
public double Humidity { get; set; }

/// <summary>
/// 室相対湿度 [-]
/// </summary>
public double RHumidity { get; set; }

/// <summary>
/// 平均表面温度 [℃]
/// </summary>
public double MRT { get; set; }

/// <summary>
/// 換気流入熱量の和[W]
/// </summary>
public double Qvent { get; set; }

/// <summary>
/// 換気流入水分量の和[kg/s]
/// </summary>
public double Jvent { get; set; }

/// <summary>
/// エアコン処理熱量の和[W]
/// </summary>
public double Qac { get; internal set; }

/// <summary>
/// エアコン処理水分量の和[kg/s]
/// </summary>
public double Jac { get; internal set; }

/// <summary>
/// 窓から取得する直達日射(放射成分)[W]
/// 単位注意(窓面積を乗じた和)
/// </summary>
public double DirectSolarRadiationTotal { get; set; }

/// <summary>
/// 窓から取得する直達日射(対流成分)[W]
/// 単位注意(窓面積を乗じた和)
/// </summary>
public double DirectSolarConvectionTotal { get; set; }

public double SkySolarRadiationTotal { get; set; }
```

```
public double SkySolarConvectionTotal { get; set; }

public double ReflectSolarRadiationTotal { get; set; }

public double ReflectSolarConvectionTotal { get;
set; }

/// <summary>
/// 内部発熱量(顕熱,対流成分)の和[W]
/// </summary>
public double SensibleHeatGenerationTotalConvection
{ get; internal set; }

/// <summary>
/// 内部発熱量(顕熱,放射成分)の和[W]
/// </summary>
public double SensibleHeatGenerationTotalRadiation
{ get; internal set; }

/// <summary>
/// 内部発熱量(潜熱)の和[kg/s]
/// </summary>
public double VaporGenerationTotal { get; internal
set; }

/// <summary>
/// 壁などの面積当たりに配分される内部発熱(放射成分)を
返す [W/m2]
/// </summary>
public double SensibleHeatGenerationRadiationToWall
{
 get
 {
  if (TotalArea > 0.0)
   return SensibleHeatGenerationTotalRadiation /
   TotalArea;
  else
   return 0.0;
 }
}

/// <summary>
/// 床以外への単位面積当たりの日射配分量(放射成分)を返
す[W/m2]
/// </summary>
public double SolarRadiationToWall
{
 get {
  if (TotalFloorArea > 0.0) //床がある
  {
   if (TotalArea > TotalFloorArea) //壁もある
    return 0.5 * (DirectSolarRadiationTotal +
    SkySolarRadiationTotal +
    ReflectSolarRadiationTotal) / (TotalArea -
    TotalFloorArea);
   else                            //壁がない(床だけ)
    return 0.0;
  }
  else  //床が定義されていない場合
  {
   if (TotalArea > 0.0)
    return (DirectSolarRadiationTotal +
    SkySolarRadiationTotal +
    ReflectSolarRadiationTotal) / TotalArea;
   else
    return 0.0;
  }
 }
}

/// <summary>
/// 床への単位面積当たりの日射配分量(放射成分)を返す
[W/m2]
/// </summary>
public double SolarRadiationToFloor
{
 get
 {
  if (TotalFloorArea > 0.0)
   return 0.5 * (DirectSolarRadiationTotal +
   SkySolarRadiationTotal +
   ReflectSolarRadiationTotal) / TotalFloorArea;
  else
   return 0.0;
 }
}

public int WindowOpen { get; set; }

public int ACONOFF { get; set; }

#endregion

/// <summary>
/// 室の計算時間間隔[s]
/// </summary>
public override long Dt { get; set; }

/// <summary>
/// 1minにおける室の計算回数
/// </summary>
private long IterationNum;


public void SetFaceConnector(WallFaceConnector
connectingface, int facetype)
{
 ConnectingFace face = new
 ConnectingFace(connectingface, facetype);
 ListConnectingFace.Add(face);
}

public void
SetVentilation(VentilationHeatTransferBlock
ventilation)
{
 ListVentilation.Add(ventilation);
}

public void SetHeatGenerator(HeatGenerator
heatgenerator)
{
 ListHeatGenerator.Add(heatgenerator);
}

public void SetPerson(PersonBlock person)
{
 Person = person;
}

public void SetAirConditioner(AirConditionerBlock ac)
{
 AC = ac;
```

```
}

/// <summary>
/// 面の角度を受け取り,当該面に入射する日射量,夜間放射量を返す
/// </summary>
public void GetSolar(double costheta, double facecosh, out double directsolar, out double skysolar, out double reflectsolar, out double longwave)
{
 directsolar = 0.0;
 skysolar = 0.0;
 reflectsolar = 0.0;
 longwave = 0.0;
}

/// <summary>
/// 面の角度を受け取り,太陽方向となす角のcosを返す
/// </summary>
/// <param name="facewz">面のcos(傾斜角)</param>
/// <param name="faceww">面のsin(傾斜角)×sin(方位角)</param>
/// <param name="faceww">面のsin(傾斜角)×cos(方位角)</param>
public double GetCosTheta(double facewz, double faceww, double facews)
{
 return 0.0;
}

public override void Init(SimulationContext context)
{

 /// 顕熱容量
 if (Cs <= 0.0) Cs = c_air * rho_air * Volume;
 if (Cl <= 0.0) Cl = rho_air * Volume;

 Qvent = 0.0;
 Jvent = 0.0;
 Qac = 0.0;
 Jac = 0.0;
 DirectSolarRadiationTotal = 0.0;
 DirectSolarConvectionTotal = 0.0;
 SkySolarRadiationTotal = 0.0;
 SkySolarConvectionTotal = 0.0;
 ReflectSolarRadiationTotal = 0.0;
 ReflectSolarConvectionTotal = 0.0;
 SensibleHeatGenerationTotalConvection = 0.0;
 SensibleHeatGenerationTotalRadiation = 0.0;
 VaporGenerationTotal = 0.0;


 ///
 ///対流・総合熱伝達相手先の集計
 ///
 double alphaca = 0.0;          ///ΣαcA
 foreach (ConnectingFace face in ListConnectingFace)
 {
  alphaca += face.Face.Area * face.Face.AlphaC;
  TotalArea += face.Face.Area;

  if (face.FaceType == 1)
   TotalFloorArea += face.Face.Area;
 }

 ///
 ///換気相手先(自分が下流のもの)の集計
 ///
 double crhov = 0.0;          ///ΣcρV
 double rhov = 0.0;           ///ΣρV

 crhov += c_air * rho_air * Math.Max(VentNumClose, VentNumOpen) * Volume / 3600.0;   ///自然換気分
 rhov += rho_air * Math.Max(VentNumClose, VentNumOpen) * Volume / 3600.0;             ///自然換気分

 foreach (VentilationHeatTransferBlock ventilation in ListVentilation)
 {
  crhov += c_air * rho_air * ventilation.AirVolume;
  rhov += rho_air * ventilation.AirVolume;
 }

 double t1 = 60.0;    ///熱の式から求まる時間間隔   仮定値(10s)
 double t2 = 60.0;    ///湿気の式から求まる時間間隔 仮定値(10s)
 if (alphaca + crhov > 0.0)
  t1 = Cs / (alphaca + crhov);

 if (rhov > 0.0)
  t2 = Cl / rhov;

 double t;
 if (t1 > t2) t = t2;
 else t = t1;

 Dt = (long)(t * 1000.0);    ///[ms]単位で保持する

 if (Dt == 0)
 {
  System.Environment.Exit(1);
 }

 Dt = base.CalcDt(Dt);
 IterationNum = 60000 / Dt;     ///1minの反復回数

}

/// <summary>
/// 換気熱・水分移動量を受け取る(換気扇によるもののみ)
/// <param name="qv">換気熱移動量[W]</param>
/// <param name="jv">換気水分移動量[kg/s]</param>
/// </summary>
public void AddVent(double qv, double jv)
{
 Qvent += qv;
 Jvent += jv;
}

/// <summary>
/// エアコンの処理熱量を受け取る
/// </summary>
/// <param name="qac">エアコン処理顕熱量[W] 冷房が正</param>
/// <param name="jac">エアコン処理水分量[kg/s] 除湿が正</param>
public void AddAC(double qac, double jac)
{
 Qac += qac;
 Jac += jac;
}

/// <summary>
```

```
/// 反復前の毎回の初期化
/// </summary>
/// <param name="context"></param>
public override void IterationInit(SimulationContext
context)
{
 ///
 /// 内部発熱量の総和を求める[W]
 ///
 double qgenc = 0.0;
 double qgenr = 0.0;
 double jgen = 0.0;    ///[kg/s]
 for (int i = 0; i < ListHeatGenerator.Count; i++)
 {
  ListHeatGenerator[i].GetHeat(context, out qgenc,
  out qgenr, out jgen);

  SensibleHeatGenerationTotalConvection += qgenc;
  SensibleHeatGenerationTotalRadiation += qgenr;
  VaporGenerationTotal += jgen;
 }

 ///
 ///在室者
 ///
 if (Person != null)
 {
  Person.GetHeat(context, out qgenc, out qgenr, out
  jgen);
  SensibleHeatGenerationTotalConvection += qgenc;
  SensibleHeatGenerationTotalRadiation += qgenr;
  VaporGenerationTotal += jgen;
 }
 ///
 ///エアコン判定
 ///
 if (AC != null)
 {
  if (AC.WorkType == 0)     ///スケジュールどおり
  {
   AC.SetOperation(context);
   if (AC.currentOperation)
    this.ACONOFF = 1;
   else
    this.ACONOFF = 0;
  }
  else if (AC.WorkType == 1)     ///スケジュール+室温
  参照
  {
   AC.SetOperation(context, this.Temp);
   if (AC.currentOperation)
    this.ACONOFF = 1;
   else
    this.ACONOFF = 0;
  }
  else if (AC.WorkType == 2)     ///人体判定結果
  {
   if (Person != null)
   {
    int windowopen_new = 0;
    int aconoff_new = 0;
    Person.ACONOFF(context, Temp, Outside.Temp,
    WindowOpen, ACONOFF,
     out windowopen_new, out aconoff_new);

    WindowOpen = windowopen_new;
    ACONOFF = aconoff_new;
   }
   AC.SetOperation(this.ACONOFF);
  }

 }
}

public override void Run(SimulationContext context)
{

 double qc = 0.0;           //対流熱伝達量[W]
 foreach (ConnectingFace face in ListConnectingFace)
 {
  qc -= face.Face.GetHeat() * face.Face.Area;
  ///GetHeatはスペースから面に移動する量が正[W/m2]
 }
 double qs = DirectSolarConvectionTotal +
 SkySolarConvectionTotal +
 ReflectSolarConvectionTotal;  //日射の対流成分[W]
 double qg = SensibleHeatGenerationTotalConvection;

 ///自然換気(窓開け判定結果を反映)
 double qnv = 0.0;
 double jnv = 0.0;
 double ventnum = 0.0;
 if (WindowOpen == 0)
  ventnum = VentNumClose;
 else
  ventnum = VentNumOpen;

 qnv += 1005.0 * rho_air * ventnum * Volume / 3600.0 *
 (Outside.Temp - this.Temp);
 jnv += rho_air * ventnum * Volume / 3600.0 *
 (Outside.Humidity - this.Humidity);
 Qvent += qnv;
 Jvent += jnv;

 double TtmpOld = Temp;     //暫定温度
 double TtmpNew = Temp;

 for (long n = 0; n < IterationNum; n++)    ///この室
 の反復
 {
  TtmpNew = TtmpOld + (qc + Qvent + qs + qg - Qac) *
  (Dt / 1000.0) / Cs;    //Dtは[ms]なので注意
  TtmpOld = TtmpNew;
 }


 double jc = 0.0;          //表面水分伝達量

 double XtmpOld = Humidity;   //暫定絶対湿度
 double XtmpNew = Humidity;   //暫定絶対湿度

 double jgen = VaporGenerationTotal;    ///内部発熱(潜
 熱) [kg/s]

 for (long n = 0; n < IterationNum; n++)
 {
  XtmpNew = XtmpOld + (jc + Jvent + jgen - Jac) * (Dt
  / 1000.0) / Cl;
  XtmpOld = XtmpNew;
 }

 ///
 ///反復終了
 ///
```

```
  Temp = TtmpNew;
  Humidity = XtmpNew;
  RHumidity = Utility.Util.CalcHumidity(Temp, Humidity,
  3);

  ///相対湿度が100%超となったときの処理(100%まで下げる)
  if (RHumidity > 1.0)
  {
   Humidity = Utility.Util.CalcHumidity(Temp, 1.0, 1);
   ///相湿100%となる絶対湿度
   RHumidity = 1.0;
  }
}


public override void Commit(SimulationContext context)
{

  Qvent = 0.0;
  Jvent = 0.0;
  Qac = 0.0;
  Jac = 0.0;
  SensibleHeatGenerationTotalRadiation = 0.0;
  SensibleHeatGenerationTotalConvection = 0.0;
  VaporGenerationTotal = 0.0;

  DirectSolarRadiationTotal = 0.0;
  DirectSolarConvectionTotal = 0.0;
  SkySolarRadiationTotal = 0.0;
  SkySolarConvectionTotal = 0.0;
  ReflectSolarRadiationTotal = 0.0;
  ReflectSolarConvectionTotal = 0.0;

  ///
  ///表面のMRT(Σ θ jAj)の計算 次時刻の放射熱伝達計算に
  用いる
  ///
  MRT = 0.0;
  for ( int n =0 ; n <ListConnectingFace.Count; n++)
  {
   double temp =
   ListConnectingFace[n].Face.SurfaceTemp;

   MRT += temp * ListConnectingFace[n].Face.Area;
  }
  MRT /= TotalArea;

  base.Commit(context);
}

/// <summary>
/// ダミー
/// </summary>
/// <param name="facewz"></param>
/// <param name="faceww"></param>
/// <param name="facews"></param>
/// <returns></returns>
public double GeCosTheta(double facewz, double faceww,
double facews)
{ return 0.0; }

public double GetSAT(double solarabsorptionrate,
double longwaveemissionrate, double alphat, double
costheta, double facecosh)
{ return 0.0; }

public double GetSAT(double longwaveemissionrate,
   double alphat, double costheta, double facecosh)
   { return 0.0; }
  }
}
```

1.10 WallBlock.cs

```
using HLFX.Utility;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace HLFX.Block
{
 /// <summary>
 /// 壁
 /// </summary>
 public class WallBlock : BaseBlock, ISurface
 {
  /// <summary>
  /// 面積[m^2]
  /// </summary>
  public double Area { get; set; }

  /// <summary>
  /// A側スペースにとっての床フラグ(1なら床)
  /// </summary>
  public int FloorofSpaceA { get; set; }

  /// <summary>
  /// 材料厚さ[m]
  /// </summary>
  public double[] Thickness { get; }

  /// <summary>
  /// 材料
  /// </summary>
  public Material[] Material { get; }

  /// <summary>
  /// 層構成の向き 1:層構成のI側が壁のA側, 2:逆
  /// </summary>
  public int MatDirection { get; }

  /// <summary>
  /// 表面A
  /// </summary>
  public WallFaceConnector FaceA { get; private set; }

  /// <summary>
  /// 表面B
  /// </summary>
  public WallFaceConnector FaceB { get; private set; }

  public double Costheta { get; set; }

  private int NodeNumber;

  private double[] q_WL;
  private double[] J_WL;
  private double[] gamma;
```

```
private double[] gamma_x;
private double[] d;
private double[] T;                //節点温度
private double[] Tnew;             //新節点温度
private double[] X;                //節点湿度
private double[] Xnew;             //新節点湿度

private double[] NodeR;            //節点間熱抵抗 節点
[0]と節点[1]の間を[0]とする
private double[] NodeC;            //節点熱容量

/// <summary>
/// 節点温度 I→J
/// </summary>
public double[] Temp { get { return T; } }

/// <summary>
/// 節点湿度 I→J
/// </summary>
public double[] Humidity { get { return X; } }

/// <summary>
/// A側表面温度[℃]
/// </summary>
public double TempA
{
 get
 {
  if (MatDirection == 1)
   return T[0];
  else
   return T[NodeNumber - 1];
 }
}

/// <summary>
/// B側表面温度[℃]
/// </summary>
public double TempB
{
 get
 {
  if (MatDirection == 1)
   return T[NodeNumber - 1];
  else
   return T[0];
 }
}

/// <summary>
/// A側表面絶対湿度[kg/kg']
/// </summary>
public double HumidityA
{
 get {return X[0];}
}

/// <summary>
/// B側表面絶対湿度[kg/kg']
/// </summary>
public double HumidityB
{
 get {return X[1];}
}

/// <summary>
/// A側表面熱取得量[J]
/// </summary>
public double dU_tick_A { get; private set; }

/// <summary>
/// B側表面熱取得量[J]
/// </summary>
public double dU_tick_B { get; private set; }

/// <summary>
/// 壁の計算時間間隔[s]
/// </summary>
public override long Dt{ get; set; }

/// <summary>
/// 1minにおける壁の計算回数
/// </summary>
private long IterationNum;

public override bool IsComposite
{
 get
 {
  return true;
 }
}

public override IConnector[] GetConnectors()
{
 return new IConnector[] { FaceA, FaceB };
}

/// <summary>
/// コンストラクタ
/// </summary>
/// <param name="spaceA">A側居室</param>
/// <param name="spaceB">B側居室または外部</param>
/// <param name="area">面積[m^2]</param>
/// <param name="floorofspaceA">A側居室にとっての床フ
ラグ</param>
/// <param name="materialList">材料の配列</param>
/// <param name="thicknessList">材料厚さ[m]の配列
</param>
/// <param name="matdirection">層構成の向き(1:層構成I
側が壁のA側,2:逆)</param>
/// <param name="alphacA">A側対流熱伝達率
[W/m2K]</param>
/// <param name="alpharA">A側放射熱伝達率
[W/m2K]</param>
/// <param name="solarabsorptionrateA">A側日射吸収率
[-]</param>
/// <param name="longwaveemissionrateA">A側長波放射率
[-]</param>
/// <param name="alphacB">B側対流熱伝達率
[W/m2K]</param>
/// <param name="alpharB">B側放射熱伝達率
[W/m2K]</param>
/// <param name="solarabsorptionrateB">B側日射吸収率
[-]</param>
/// <param name="longwaveemissionrateB">B側長波放射率
[-]</param>
/// <param name="faceh">面傾斜角[deg]</param>
/// <param name="facea">面方位角[deg]</param>
public WallBlock(string name, ISpaceBlock spaceA,
ISpaceBlock spaceB, double area, int floorofspaceA,
Material[] materialList, double[] thicknessList, int
matdirection,
  double alphacA, double alpharA, double
```

```
 solarabsorptionrateA, double longwaveemissionrateA,
 double alphacB, double alpharB, double
 solarabsorptionrateB, double longwaveemissionrateB,
 double faceh, double facea)
 : base()
{
 this.Name = name;
 this.Area = area;
 this.FloorofSpaceA = floorofspaceA;
 this.Thickness = thicknessList;
 this.Material = materialList;
 this.MatDirection = matdirection;
 this.FaceA = new WallFaceConnector(name+"A", spaceA,
 this, 1, 0, area, alphacA, alpharA,
 solarabsorptionrateA, longwaveemissionrateA, faceh,
 facea);
 this.FaceB = new WallFaceConnector(name+"B", spaceB,
 this, 2, 6, area, alphacB, alpharB,
 solarabsorptionrateB, longwaveemissionrateB, faceh,
 facea);

 //spaceAに登録
 if (spaceA is RoomBlock)
 {
  spaceA.SetFaceConnector(this.FaceA, floorofspaceA);
 }

 //spaceBに登録
 if (spaceB is RoomBlock)
 {
  spaceB.SetFaceConnector(this.FaceB, 0);
 }

 //作業領域作成
 int matnum = materialList.Length;      //材料数
 NodeNumber = materialList.Length;      //原則として節
 点数=材料数とする，端部材料節点は材料表面にあり，内部
 材料節点は材料中央にあるものとする
 if (matnum == 1) NodeNumber = 2;       //材料数=1の場
 合は両側に節点を置く

 d = new double[matnum];                //各材料厚さ
 for (int i = 0; i < matnum; i++)
 {
  d[i] = thicknessList[i];
 }

 double[] lambda = new double[matnum];   //各材料熱伝
 導率
 double[] cr = new double[matnum];       //各材料容積
 比熱
 double[] mr = new double[matnum];       //各材料熱抵
 抗
 double[] mc = new double[matnum];       //各材料熱容
 量
 for (int i = 0; i < matnum; i++)
 {
  lambda[i] = materialList[i].熱伝導率;      //空気層
  の場合は0なので注意
  cr[i] = materialList[i].容積比熱;
  //[J/m3K]

  if (lambda[i] > 0.0)
   mr[i] = d[i] / lambda[i];
  else
   mr[i] = materialList[i].空気層熱抵抗;
  mc[i] = cr[i] * d[i];
 }

 NodeR = new double[NodeNumber - 1];
 NodeC = new double[NodeNumber];


 if (matnum == 1)                        ///材料数1(節
 点数 = 2)
 {
  NodeR[0] = mr[0];
  NodeC[0] = NodeC[1] = 0.5 * mc[0];
 }
 else if (matnum == 2)                   ///材料数2(節
 点数 = 2)
 {
  NodeR[0] = mr[0] + mr[1];
  NodeC[0] = mc[0];
  NodeC[1] = mc[1];
 }
 else                                    ///材料数3以上(節
 点数 = 材料数)
 {
  for (int i = 0; i < NodeNumber - 1; i++)
  {
   if (i == 0)                                    ///I
   端節点と2番目節点(節点番号0, 1)
    NodeR[i] = mr[i] + 0.5 * mr[i + 1];
   else if (i == NodeNumber - 2)                  ///J
   端の手前節点とJ端節点(節点番号NodeNumber-2,
   NodeNumber-1)
    NodeR[i] = 0.5 * mr[i] + mr[i + 1];
   else                                           ///
   中間節点
    NodeR[i] = 0.5 * mr[i] + 0.5 * mr[i + 1];

   NodeC[i] = mc[i];
  }
  NodeC[NodeNumber - 1] = mc[NodeNumber - 1];

 }

 ///収束条件のチェック
 double t = 60.0;            /// 20s
 double dt_tmp = 0.0;

 for (int i = 0; i < NodeNumber; i++)
 {
  if (i == 0)
  {
   if (MatDirection == 1)
    dt_tmp = NodeC[i] / (alphacA + alpharA + 1.0 /
    NodeR[i]);   ///材料I側=壁A側
   else
    dt_tmp = NodeC[i] / (alphacB + alpharB + 1.0 /
    NodeR[i]);   ///材料I側=壁B側
  }
  else if (i == NodeNumber - 1)
  {
   if (MatDirection == 1)
    dt_tmp = NodeC[i] / (alphacB + alpharB + 1.0 /
    NodeR[i - 1]);   ///材料J側=壁B側
   else
    dt_tmp = NodeC[i] / (alphacA + alpharA + 1.0 /
    NodeR[i - 1]);   ///材料J側=壁A側
  }
  else
```

```
    dt_tmp = NodeC[i] / (1.0 / NodeR[i - 1] + 1.0 /
    NodeR[i]);

   if (dt_tmp < t) t = dt_tmp;      ///時間間隔制限が最
   も厳しいものを保持

  }
  Dt = (long) (t * 1000.0);     ///[ms]単位で保持する

  if (Dt == 0)
  {
   System.Environment.Exit(1);
  }

  Dt = base.CalcDt(Dt);
  IterationNum = 60000 / Dt;      ///20sを進めるための反
  復回数


  q_WL = new double[matnum + 1];     //
  J_WL = new double[matnum + 1];     //
  gamma = new double[matnum];        //
  gamma_x = new double[matnum];      //
  T = new double[NodeNumber];          //節点温度
  Tnew = new double[NodeNumber];       //新節点温度
  X = new double[NodeNumber];          //節点絶対湿度
  Xnew = new double[NodeNumber];       //新節点絶対湿度


  double initialtemp = 20.0;
  double initialhumidity = 7.354 / 1000.0;
  for (int i = 0; i < NodeNumber; i++)
  {
   T[i] = initialtemp;
   Tnew[i] = initialtemp;
   X[i] = initialhumidity;
   Xnew[i] = initialhumidity;
  }
}



/// <summary>
/// 熱移動
/// </summary>
/// <param name="dU">熱移動量[J]</param>
public void AddHeatA(double dU, IConnector src,
IConnector kernel)
{
 dU_tick += dU;
}


public override void Run(SimulationContext context)
{
 const double k = 100000.0;
 const double v = 0.0;


 //熱流・水分流
 double[] TtmpOld = new double[NodeNumber];      //1min
 進行するまでの(仮)更新後温度
 double[] TtmpNew = new double[NodeNumber];      //1min
 進行するまでの(仮)更新後温度

 //I側・J側表面
 double qi = 0.0;            //i側隣接節点からの熱
 [W/m2]
 double qj = 0.0;            //j側隣接節点からの熱
 [W/m2]
 double qspci = 0.0;         //i側スペースからの対流熱
 流[W/m2]
 double qspcj = 0.0;         //j側スペースからの対流熱
 流[W/m2]
 double qradi = 0.0;         //i側スペースからの放射熱
 流[W/m2]
 double qradj = 0.0;         //j側スペースからの放射熱
 流[W/m2]
 double qgeni = 0.0;         //i側スペースの内部発熱(放
 射成分)[W/m2]
 double qgenj = 0.0;         //j側スペースの内部発熱(放
 射成分)[W/m2]
 double qsoli = 0.0;         //i側スペースの日射配分(放
 射成分)[W/m2]
 double qsolj = 0.0;         //j側スペースの日射配分(放
 射成分)[W/m2]

 for (int i = 0; i < NodeNumber; i++)
  TtmpOld[i] = T[i];

 if (MatDirection == 1)         ///材料i側=面A側
 {
  qspci = FaceA.GetHeat();     ///先頭材料はA側
  qspcj = FaceB.GetHeat();     ///末尾材料はB側

  if (FaceA.Space is FixedConditionRoomBlock)
  {
  }
  else if (FaceA.Space is RoomBlock)
  {
   qradi = FaceA.AlphaR * (FaceA.Space.MRT -
   TtmpOld[0]);                  ///先頭材料への放射熱
   qgeni =
   FaceA.Space.SensibleHeatGenerationRadiationToWall;

   if (this.FloorofSpaceA == 1)
    qsoli = FaceA.Space.SolarRadiationToFloor;
   else
    qsoli = FaceA.Space.SolarRadiationToWall;
  }

  if (FaceB.Space is FixedConditionRoomBlock)
  {
  }
  else if (FaceB.Space is RoomBlock)
  {
   qradj = FaceB.AlphaR * (FaceB.Space.MRT -
   TtmpOld[NodeNumber -1]);    ///末尾材料への放射熱
   qgenj =
   FaceB.Space.SensibleHeatGenerationRadiationToWall;
   qsolj = FaceB.Space.SolarRadiationToWall;
  }
 }
 else
 {
  qspci = FaceB.GetHeat();     ///先頭材料はB側
  qspcj = FaceA.GetHeat();     ///末尾材料はA側

  if (FaceA.Space is FixedConditionRoomBlock)
  {
  }
  else if (FaceA.Space is RoomBlock)
  {
   qradj = FaceA.AlphaR * (FaceA.Space.MRT -
```

```
    TtmpOld[NodeNumber - 1]);   ///末尾材料への放射熱
    qgenj =
    FaceA.Space.SensibleHeatGenerationRadiationToWall;

    if (this.FloorofSpaceA == 1)
      qsolj = FaceA.Space.SolarRadiationToFloor;
    else
      qsolj = FaceA.Space.SolarRadiationToWall;
  }

  if (FaceB.Space is FixedConditionRoomBlock)
  {
  }
  else if (FaceB.Space is RoomBlock)
  {
    qradi = FaceB.AlphaR * (FaceB.Space.MRT -
    TtmpOld[0]);                ///先頭材料への放射熱
    qgeni =
    FaceB.Space.SensibleHeatGenerationRadiationToWall;
    qsoli = FaceB.Space.SolarRadiationToWall;
  }

}


for (long n = 0; n < IterationNum; n++)         ///こ
の壁の反復
{
  for (int i = 0; i < NodeNumber; i++)        ///節点
  巡回
  {
    //I側隣接点からの熱流
    if (i == 0)
      qi = qspci + qradi + qsoli + qgeni;
    else
      qi = -qj;

    //J側隣接点からの熱流
    if (i == NodeNumber - 1)
      qj = qspcj+ qradj + qsolj + qgenj;
    else
      qj = 1.0 / NodeR[i] * (TtmpOld[i + 1] -
      TtmpOld[i]);  //W/m2

    /// 温度更新
    TtmpNew[i] = TtmpOld[i] + (qi + qj) * (Dt /
    1000.0) / NodeC[i];    //Dtは[ms]なので注意
  }

  for (int i = 0; i < NodeNumber; i++)
    TtmpOld[i] = TtmpNew[i];

}

q_WL[0] = FaceA.GetHeat();  //TODO: q_r->me 相当の処
理の実装
J_WL[0] = FaceA.GetMoisture();  //TODO: q_r->me 相当
の処理の実装

q_WL[1] = (T[0] - T[1]) / (d[0] / gamma[0] + 0.5 *
d[1] / gamma[1]);
J_WL[1] = (X[0] - X[1]) / (d[0] / gamma_x[0] + 0.5 *
d[1] / gamma_x[1]);

for (int i = 2; i < n - 1; i++)
{
  q_WL[i] = (TWL[i - 1] - TWL[i]) / (0.5 * (d[i - 1]
    / gamma[i - 1] + d[i] / gamma[i]));
    J_WL[i] = (XWL[i - 1] - XWL[i]) / (0.5 * (d[i - 1]
    / gamma_x[i - 1] + d[i] / gamma_x[i]));
  }

  q_WL[n - 1] = (TWL[n - 2] - TWL[n - 1]) / (0.5 * d[n
  - 2] / gamma[n - 1] + d[n - 1] / gamma[n - 1]);
  J_WL[n - 1] = (XWL[n - 2] - XWL[n - 1]) / (0.5 * d[n
  - 2] / gamma_x[n - 1] + d[n - 1] / gamma_x[n - 1]);

  q_WL[n] = -FaceB.GetHeat();
  J_WL[n] = -FaceB.GetMoisture();


  //温湿度更新
  for (int i = 0; i < d.Length; i++)
  {
    var AT = v * d[i] * Area;
    var AX = (0.5 * 1.2 + k) * d[i] * Area;
    var BT = (1000.0 * 700.0 + 2500.0 * v) * d[i] *
    Area;
    var BX = 2500.0 * k * d[i] * Area;

    var dT = (AX * (q_WL[i] - q_WL[i + 1]) + BX *
    (J_WL[i] - J_WL[i + 1])) / (AX * BT - AT * BX) *
    Area;
    if (!Double.IsNaN(dT)) T[i] += dT;

    var dX = (AT * (q_WL[i] - q_WL[i + 1]) + BT *
    (J_WL[i] - J_WL[i + 1])) / (AX * BT - AT * BX) *
    Area;
    if (!Double.IsNaN(dX)) X[i] += dX;
  }

  ///
  ///反復終了
  ///
  for (int i = 0; i < NodeNumber; i++)
    Tnew[i] = TtmpNew[i];


  base.Run(context);
}

public override void Commit(SimulationContext context)
{

  base.Commit(context);

  for (int i = 0; i < NodeNumber; i++)
    T[i] = Tnew[i];

}

public override void Init(SimulationContext context)
{
  ///base.Dt = context.WallDt;
}

public void AddHeatA()
{

}
public void AddHeatB()
{

}
```

```
    public void AddMoistureA()
    {

    }
    public void AddMoistureB()
    {

    }

  }
}
```

1.11 WindowBlock.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace HLFX.Block
{
  /// <summary>
  /// 窓
  /// </summary>
  public class WindowBlock: BaseBlock, ISurface, IGlass
  {

    #region 特性

    /// <summary>
    /// 窓面積[m2]
    /// </summary>
    public double Area { get { return Width * Height; } }

    /// <summary>
    /// 幅[m2]
    /// </summary>
    public double Width { get; private set; }

    /// <summary>
    /// 高さ[m2]
    /// </summary>
    public double Height { get; private set; }

    /// <summary>
    /// 熱貫流率
    /// </summary>
    public double UValue { get; private set; }

    /// <summary>
    /// ηR
    /// </summary>
    public double EtaR { get; set; }

    /// <summary>
    /// ηC
    /// </summary>
    public double EtaC { get; set; }

    /// <summary>
    /// ガラス厚さ[m] 熱容量算定用
    /// </summary>
    public double Thickness { get; private set; }

    /// <summary>
    /// 表面A
    /// </summary>
    public WallFaceConnector FaceA { get; private set; }

    /// <summary>
    /// 表面B
    /// </summary>
    public WallFaceConnector FaceB { get; private set; }

    /// <summary>
    /// 太陽方向となす角
    /// </summary>
    public double Costheta { get; set; }

    /// <summary>
    /// 窓の計算時間間隔[ms]
    /// </summary>
    public override long Dt { get; set; }

    /// <summary>
    /// 1minにおける壁の計算回数
    /// </summary>
    private long IterationNum;

    private int NodeNumber = 2;
    private double[] T;           //節点温度
    private double[] Tnew;        //新節点温度
    private double[] X;           //節点湿度
    private double[] Xnew;        //新節点湿度
    private double NodeR;         //節点間熱抵抗
    private double[] NodeC;       //節点熱容量

    #endregion

    #region 状態値

    /// <summary>
    /// A側表面温度[℃]
    /// </summary>
    public double TempA
    {
      get
      {
        return T[0];    //窓は必ずA側をT[0],B側をT[1]とする
      }
    }

    /// <summary>
    /// B側表面温度[℃]
    /// </summary>
    public double TempB
    {
      get
      {
        return T[1];
      }
    }

    /// <summary>
    /// A側表面絶対湿度[kg/kg']
    /// </summary>
    public double HumidityA { get; private set; }

    /// <summary>
```

```
/// B側表面絶対湿度[kg/kg']
/// </summary>
public double HumidityB { get; private set; }

#endregion

public override bool IsComposite
{
 get
 {
  return true;
 }
}

public override IConnector[] GetConnectors()
{
 return new IConnector[] { FaceA, FaceB };
}

/// <summary>
/// コンストラクタ
/// </summary>
/// <param name="area">面積[m^2]</param>
/// <param name="windowProperty">窓仕様No</param>
/// <param name="width">幅[m]</param>
/// <param name="height">高さ[m]</param>
/// <param name="faceh">面傾斜角[deg]</param>
/// <param name="facea">面方位角[deg]</param>
public WindowBlock(string name, ISpaceBlock spaceA,
ISpaceBlock spaceB, double width, double height,
 double uvalue, double etar, double etac, double
 thickness, double faceh, double facea) : base()
{
 this.Name = name;
 this.Width = width;
 this.Height = height;
 this.UValue = uvalue;
 this.EtaR = etar;
 this.EtaC = etac;
 this.Thickness = thickness;

 double alphacA = 4.4;
 double alpharA = 4.7;
 double alphacB = 20.3;
 double alpharB = 4.7;
 double solarabsorptionrate = 0.0;
 double longwaveemissionrate = 0.9;

 if (spaceB is RoomBlock)
 {
  alphacB = alphacA;
 }

 this.FaceA = new WallFaceConnector(name + "A",
 spaceA, this, 1, 0, this.Area,
  alphacA, alpharA, solarabsorptionrate,
  longwaveemissionrate, faceh, facea);
 this.FaceB = new WallFaceConnector(name + "B",
 spaceB, this, 2, 0, this.Area,
  alphacB, alpharB, solarabsorptionrate,
  longwaveemissionrate, faceh, facea);

 //spaceAに登録
 if (spaceA is RoomBlock)
 {
  spaceA.SetFaceConnector(this.FaceA, 0);
 }

 //spaceBに登録
 if (spaceB is RoomBlock)
 {
  spaceB.SetFaceConnector(this.FaceB, 0);
 }

 //作業領域作成
 double cr = 2268 * 1000.0;          ///ガラス容積比熱
 [J/m3K]

 NodeR = 1.0 / uvalue - 1.0 / (alphacA + alpharA) -
 1.0 / (alphacB + alpharB);
 NodeC = new double[NodeNumber];
 for (int i = 0; i < NodeNumber; i++)
 {
  NodeC[i] = cr * 0.5 * thickness;
 }


 ///収束条件のチェック
 double t = 60.0;           ///最大20s
 double dt_tmp = 0.0;       ///

 dt_tmp = NodeC[1] / (alphacB + alpharB + 1.0 /
 NodeR);   ///B側が条件的に厳しい
 if (dt_tmp < t) t = dt_tmp;
 Dt = (long)(t * 1000.0);   ///[ms]単位で保持する

 if (Dt == 0)
 {
  System.Environment.Exit(1);
 }


 Dt = base.CalcDt(Dt);
 IterationNum = 60000 / Dt;     ///20sを進めるための反
 復回数

 T = new double[NodeNumber];
 Tnew = new double[NodeNumber];
 X = new double[NodeNumber];
 Xnew = new double[NodeNumber];

 double initialtemp = 20.0;
 double initialhumidity = 0.0;
 for (int i = 0; i < NodeNumber; i++)
 {
  T[i] = initialtemp;
  Tnew[i] = initialtemp;
  X[i] = initialhumidity;
  Xnew[i] = initialhumidity;
 }

}

/// <summary>
/// 反復前の毎回の初期化(室への日射導入)
/// </summary>
/// <param name="context"></param>
public override void IterationInit(SimulationContext
context)
{
 ///
 ///透過日射
 ///
 if (FaceB.Space is OutsideBlock)
```

```
  {
   double costheta = FaceB.CosTheta;
   double sd = FaceB.DirectSolar;
   double ss = FaceB.SkySolar;
   double sr = FaceB.ReflectSolar;

   ISpaceBlock SpaceA = FaceA.Space;            ///ス
   ペースの取得
   double afd = AngleFactor(costheta) / 0.88;   //入射
   角特性(直達)      基準化
   double afs = AngleFactor() / 0.88;           //入射
   角特性(天空反射) 基準化

   ///
   ///室へ日射を与える
   ///

   SpaceA.DirectSolarRadiationTotal += sd * afd * EtaR
   * this.Area;
   SpaceA.DirectSolarConvectionTotal += sd * afd *
   EtaC * this.Area;
   SpaceA.SkySolarRadiationTotal += ss * afs * EtaR *
   this.Area;
   SpaceA.SkySolarConvectionTotal += ss * afs * EtaC *
   this.Area;
   SpaceA.ReflectSolarRadiationTotal += sr * afs *
   EtaR * this.Area;
   SpaceA.ReflectSolarConvectionTotal += sr * afs *
   EtaC * this.Area;
  }
}

public override void Run(SimulationContext context)
{

  //熱流・水分流
  double[] TtmpOld = new double[NodeNumber];     //1min
  進行するまでの(仮)更新後温度
  double[] TtmpNew = new double[NodeNumber];     //1min
  進行するまでの(仮)更新後温度

  for (int i = 0; i < NodeNumber; i++)
   TtmpOld[i] = T[i];

  //I側・J側表面
  //窓はI側=A側で固定
  double qi = 0.0;                    //i側隣接節点からの
  熱[W/m2]
  double qj = 0.0;                    //j側隣接節点からの
  熱[W/m2]
  double qradi = 0.0;
  double qradj = 0.0;
  double qgeni = 0.0;
  double qgenj = 0.0;

  double qspci = FaceA.GetHeat();         //i側スペース
  からの対流熱[W/m2] スペースから面に入る場合を正
  double qspcj = FaceB.GetHeat();         //j側スペース
  からの対流熱[W/m2] スペースから面に入る場合を正

  if (FaceA.Space is RoomBlock)
  {
   qradi = FaceA.AlphaR * (FaceA.Space.MRT -
   TtmpOld[0]); //i側スペースからの放射熱[W/m2]
   qgeni =
   FaceA.Space.SensibleHeatGenerationRadiationToWall;
  }

  if (FaceB.Space is RoomBlock)
  {
   qradj = FaceB.AlphaR * (FaceB.Space.MRT -
   TtmpOld[NodeNumber - 1]);                    //j側
   スペースからの放射熱[W/m2]
   qgenj =
   FaceB.Space.SensibleHeatGenerationRadiationToWall;
  }


  for (int i = 0; i < NodeNumber; i++)
   TtmpOld[i] = T[i];

  for (long n = 0; n < IterationNum; n++)    ///この窓
  の反復
  {
   for (int i = 0; i < NodeNumber; i++)
   {
    //I側隣接点からの熱流
    if (i == 0)
    {
     qi = qspci + qradi + qgeni;
     qj = 1.0 / NodeR * (TtmpOld[i + 1] - TtmpOld[i]);
     //W/m2
    }
    else
    {
     qi = -qj;
     qj = qspcj + qradj + qgenj;
    }

    /// 温度更新
    TtmpNew[i] = TtmpOld[i] + (qi + qj) * (Dt /
    1000.0) / NodeC[i];    //Dtは[ms]なので注意

   }
  }

  ///
  ///反復終了
  ///
  for (int i = 0; i < NodeNumber; i++)
   Tnew[i] = TtmpNew[i];


}


public override void Commit(SimulationContext context)
{
 base.Commit(context);

 for (int i = 0; i < NodeNumber; i++)
  T[i] = Tnew[i];
}

public override void Init(SimulationContext context)
{
 ///base.Dt = context.WindowDt;
}

public void AddHeatA()
{

}
```

```
    public void AddHeatB()
    {

    }

    public void AddMoistureA()
    {

    }

    public void AddMoistureB()
    {

    }

    public double AngleFactor(double costheta)
    {
      if (costheta <= 0.0)
        return 0.0;
      double a = costheta * costheta;
      return costheta * (2.392 + a * (-3.8636  + a *
      (3.7568  - 1.3952 * a)));
    }

    public double AngleFactor()
    {
      return 0.81;
    }

  }

  public interface IGlass
  {
    double EtaR { get; set; }
    double EtaC { get; set; }
    double AngleFactor(double costheta);
  }
}
```

1.12 DoorBlock.cs

```
using HLFX.Utility;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace HLFX.Block
{
  public class DoorBlock : BaseBlock, ISurface
  {
    #region 特性

    /// <summary>
    /// 面積[m2]
    /// </summary>
    public double Area { get { return Width * Height; } }

    /// <summary>
    /// 幅[m2]
    /// </summary>
    public double Width { get; private set; }

    /// <summary>
    /// 高さ[m2]
    /// </summary>
    public double Height { get; private set; }

    /// <summary>
    /// 熱貫流率
    /// </summary>
    public double UValue { get; private set; }

    /// <summary>
    /// 材料
    /// 容積比熱のみ参照する
    /// </summary>
    public Material Material { get; }

    /// <summary>
    /// 厚さ[m] 熱容量算定用
    /// </summary>
    public double Thickness { get; private set; }

    #endregion

    /// <summary>
    /// 表面A
    /// </summary>
    public WallFaceConnector FaceA { get; private set; }

    /// <summary>
    /// 表面B
    /// </summary>
    public WallFaceConnector FaceB { get; private set; }

    /// <summary>
    /// ドアの計算時間間隔[ms]
    /// </summary>
    public override long Dt { get; set; }

    /// <summary>
    /// 1minにおける壁の計算回数
    /// </summary>
    private long IterationNum;

    private int NodeNumber = 2;
    private double[] T;            //節点温度
    private double[] Tnew;         //新節点温度
    private double[] X;            //節点湿度
    private double[] Xnew;         //新節点湿度
    private double NodeR;          //節点間熱抵抗
    private double[] NodeC;        //節点熱容量

    public double Costheta { get; set; }
    #region 状態値

    /// <summary>
    /// A側表面温度[℃]
    /// </summary>
    public double TempA
    {
      get
      {
        return T[0];    //窓は必ずA側をT[0],B側をT[1]とする
      }
    }

    /// <summary>
```

```
/// B側表面温度[℃]
/// </summary>
public double TempB
{
 get
 {
  return T[1];
 }
}

/// <summary>
/// A側表面絶対湿度[kg/kg']
/// </summary>
public double HumidityA { get; private set; }

/// <summary>
/// B側表面絶対湿度[kg/kg']
/// </summary>
public double HumidityB { get; private set; }

#endregion

/// <summary>
/// コンストラクタ
/// </summary>
/// <param name="area">面積[m^2]</param>
/// <param name="windowProperty">窓仕様No</param>
/// <param name="width">幅[m]</param>
/// <param name="height">高さ[m]</param>
/// <param name="faceh">面傾斜角[deg]</param>
/// <param name="facea">面方位角[deg]</param>
public DoorBlock(string name, ISpaceBlock spaceA,
ISpaceBlock spaceB, double width, double height,
 double uvalue,Material material,  double thickness,
 double faceh, double facea) : base()
{
 this.Name = name;
 this.Width = width;
 this.Height = height;
 this.UValue = uvalue;
 this.Thickness = thickness;
 this.Material = material;

 double alphacA = 4.4;
 double alphacB = 4.4;
 double alphar = 4.7;
 double solarabsorptionrate = 0.8;
 double longwaveemissionrate = 0.9;

 if (spaceB is OutsideBlock)
 {
  alphacB = 20.3;
 }

 if (spaceA is OutsideBlock)
 {
  alphacA = 20.3;
 }

 this.FaceA = new WallFaceConnector(name + "A",
 spaceA, this, 1, 0, this.Area,
  alphacA, alphar, solarabsorptionrate,
  longwaveemissionrate, faceh, facea);
 this.FaceB = new WallFaceConnector(name + "B",
 spaceB, this, 2, 0, this.Area,
  alphacB, alphar, solarabsorptionrate,
  longwaveemissionrate, faceh, facea);

 //spaceAに登録
 if (spaceA is RoomBlock)
 {
  spaceA.SetFaceConnector(this.FaceA, 0);
 }

 //spaceBに登録
 if (spaceB is RoomBlock)
 {
  spaceB.SetFaceConnector(this.FaceB, 0);
 }

 //作業領域作成
 double cr = Material.容積比熱;          ///材料容積比
 熱[J/m3K]

 NodeR = 1.0 / uvalue - 1.0 / (alphacA + alphar) - 1.0
 / (alphacB + alphar);
 NodeC = new double[NodeNumber];
 for (int i = 0; i < NodeNumber; i++)
 {
  NodeC[i] = cr * 0.5 * thickness;
 }

 ///収束条件のチェック
 double t = 60.0;           ///最大60s
 double dt_tmp = 0.0;       ///

 dt_tmp = NodeC[1] / (alphacB + alphar + 1.0 / NodeR);
 ///一方が外気だとすればB側。B側が条件的に厳しい
 if (dt_tmp < t) t = dt_tmp;
 Dt = (long)(t * 1000.0);   ///[ms]単位で保持する

 if (Dt == 0)
 {
  System.Environment.Exit(1);
 }


 Dt = base.CalcDt(Dt);
 IterationNum = 60000 / Dt;     ///1minの反復回数

 T = new double[NodeNumber];
 Tnew = new double[NodeNumber];
 X = new double[NodeNumber];
 Xnew = new double[NodeNumber];

 double initialtemp = 20.0;
 double initialhumidity = 0.0;
 for (int i = 0; i < NodeNumber; i++)
 {
  T[i] = initialtemp;
  Tnew[i] = initialtemp;
  X[i] = initialhumidity;
  Xnew[i] = initialhumidity;
 }

}

public override void Run(SimulationContext context)
{

 //熱流・水分流
 double[] TtmpOld = new double[NodeNumber];     //1min
 進行するまでの(仮)更新後温度
 double[] TtmpNew = new double[NodeNumber];     //1min
```

```
進行するまでの(仮)更新後温度

for (int i = 0; i < NodeNumber; i++)
 TtmpOld[i] = T[i];

//I側・J側表面
//窓はI側=A側で固定
double qi = 0.0;                    //i側隣接節点からの
熱[W/m2]
double qj = 0.0;                    //j側隣接節点からの
熱[W/m2]
double qradi = 0.0;
double qradj = 0.0;
double qgeni = 0.0;
double qgenj = 0.0;
double qsoli = 0.0;
double qsolj = 0.0;

double qspci = FaceA.GetHeat();         //i側スペース
からの対流または総合熱伝達[W/m2] スペースから面に入る
場合を正
double qspcj = FaceB.GetHeat();         //j側スペース
からの対流または総合熱伝達[W/m2] スペースから面に入る
場合を正

if (FaceA.Space is RoomBlock)
{
 qradi = FaceA.AlphaR * (FaceA.Space.MRT -
 TtmpOld[0]);                //i側スペースからの放射
 熱[W/m2]
 qgeni =
 FaceA.Space.SensibleHeatGenerationRadiationToWall;
 qsoli = FaceA.Space.SolarRadiationToWall;
}

if (FaceB.Space is RoomBlock)
{
 qradj = FaceB.AlphaR * (FaceB.Space.MRT -
 TtmpOld[NodeNumber - 1]);   //j側スペースからの放射
 熱[W/m2]
 qgenj =
 FaceB.Space.SensibleHeatGenerationRadiationToWall;
 qsolj = FaceB.Space.SolarRadiationToWall;
}

for (int i = 0; i < NodeNumber; i++)
 TtmpOld[i] = T[i];

for (long n = 0; n < IterationNum; n++)    ///このド
アの反復
{
 for (int i = 0; i < NodeNumber; i++)
 {
  //I側隣接点からの熱流
  if (i == 0)
  {
   qi = qspci + qradi + qgeni + qsoli;
   qj = 1.0 / NodeR * (TtmpOld[i + 1] - TtmpOld[i]);
   //W/m2
  }
  else
  {
   qi = -qj;
   qj = qspcj + qradj + qgenj+ qsolj ;
  }

  /// 温度更新
      TtmpNew[i] = TtmpOld[i] + (qi + qj) * (Dt /
      1000.0) / NodeC[i];    //Dtは[ms]なので注意

    }
   }

   ///
   ///反復終了
   ///
   for (int i = 0; i < NodeNumber; i++)
    Tnew[i] = TtmpNew[i];


  }


  public override void Commit(SimulationContext context)
  {
   base.Commit(context);

   for (int i = 0; i < NodeNumber; i++)
    T[i] = Tnew[i];
  }

  public void AddHeatA()
  {

  }

  public void AddHeatB()
  {

  }

  public void AddMoistureA()
  {

  }

  public void AddMoistureB()
  {

  }
 }
}
```

1.13 WallFaceConnector.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace HLFX.Block
{
 /// <summary>
 /// 壁面・窓面
 /// </summary>
 public class WallFaceConnector: ILayerBlock, IBlock
 {
  public bool IsComposite { get { return false; } }

  public long Dt { get; set; }
```

```
public IConnector[] GetConnectors()
{
 return null;
}

#region 接続
/// <summary>
/// 接続するオブジェクト(居室または外部環境)
/// </summary>
public ISpaceBlock Space { get; private set; }

/// <summary>
/// 接続するオブジェクト(壁/窓/ドアのいずれか)
/// </summary>
public ISurface Surface { get; private set; }

/// <summary>
/// オブジェクト(壁または窓)のどちら側の表面に接続するか(1:A側, 2:B側) 材料のI側J側ではないので注意
/// </summary>
public int SurfaceDirection { get; private set; }

#endregion

#region 特性

/// <summary>
/// 壁・窓表面積[m2]
/// </summary>
public double Area { get; private set; }

/// <summary>
/// 対流熱伝達率[W/m2K]
/// </summary>
public double AlphaC { get; private set; }

/// <summary>
/// 放射熱伝達率[W/m2K]
/// </summary>
public double AlphaR { get; private set; }

/// <summary>
/// 総合熱伝達率[W/m2K]
/// </summary>
public double AlphaT
{
 get{ return AlphaC +AlphaR; }
}

public double FaceSinH { get; set; }
public double FaceSinA { get; set; }
public double FaceCosH { get; set; }
public double FaceCosA { get; set; }
private double FaceWz;
private double FaceWw;
private double FaceWs;

///public double CosTheta { get; set; }

/// <summary>
/// 日射吸収率[-]
/// </summary>
public double SolarAbsorptionRate { get; private
set; }

/// <summary>
/// 長波長放射率[-]
/// </summary>
public double LongWaveEmissionRate { get; private
set; }

/// <summary>
/// 水分伝達率[kg/m2s(kg/kg' )]
/// </summary>
public double MoistureTransferRate { get; private
set; }

#endregion

#region 状態値

/// <summary>
/// スペース側の空気温度[℃]
/// 熱伝達計算時に呼び出される
/// 外気(総合熱伝達)の場合は表面にcosθ,入射日射量(直達・天空・反射)もセットする
/// </summary>
public double SpaceTemp
{
 get
 {
  if (Space is RoomBlock)
   return Space.Temp;
  else
  {
   double costheta = Space.GetCosTheta(FaceWz,
   FaceWw, FaceWs);
   Surface.Costheta = costheta;

   if (Surface is WallBlock || Surface is DoorBlock)
    return Space.GetSAT(SolarAbsorptionRate,
    LongWaveEmissionRate, AlphaT, costheta,
    FaceCosH);
   else if (Surface is WindowBlock)
    return Space.GetSAT(LongWaveEmissionRate, AlphaT,
    costheta, FaceCosH);
   else
    return Space.Temp;
  }
 }
}

/// <summary>
/// 絶対湿度 [kg/kg']
/// </summary>
public double SpaceHumidity
{
 get
 {
  return Space.Humidity;
 }
}

/// <summary>
/// 入射する直達日射[W/m2]
/// </summary>
public double DirectSolar { get; private set; }

/// <summary>
/// 入射する天空日射[W/m2]
/// </summary>
public double SkySolar { get; private set; }

/// <summary>
```

```
/// 入射する反射日射[W/m2]
/// </summary>
public double ReflectSolar { get; private set; }

/// <summary>
/// 放出する夜間放射[W/m2]
/// </summary>
public double LongWave { get; private set; }

/// <summary>
/// 太陽方向となす角のcos
/// </summary>
public double CosTheta { get; private set; }


/// <summary>
/// 表面温度[℃]
/// </summary>
public double SurfaceTemp
{
 get
 {
  if (SurfaceDirection == 1) return Surface.TempA;
  else return Surface.TempB;
 }
}

/// <summary>
/// 絶対湿度 [kg/kg']
/// </summary>
public double SurfaceHumidity
{
 get
 {
  if (SurfaceDirection == 1) return
  Surface.HumidityA;
  else return Surface.HumidityB;
 }
}



public string Name { get;  set; }

/// <summary>
/// 対流熱流量 単位注意[W/m2]
/// </summary>
private double Qc;

private double dJ_tick;


#endregion

/// <summary>
/// 壁計算オブジェクト
/// </summary>

private int layerIndex;

/// <summary>
/// コンストラクタ
/// </summary>
/// <param name="space">居室または外部</param>
/// <param name="surface">壁または窓</param>
/// <param name="surfacedirection">壁または窓の接続方
向(A側=1,B側=2)</param>
/// <param name="area">面積[m^2]</param>
/// <param name="alphac">対流熱伝達率[W/m2K]</param>
/// <param name="alphar">放射熱伝達率[W/m2K]</param>
/// <param name="solarabsorptionrate">日射吸収率
[-]</param>
/// <param name="longwaveemissionrate">長波放射率
[-]</param>
/// <param name="layerIndex">壁計算オブジェクトにおけ
る層インデックス</param>
/// <param name="faceh">面傾斜角[deg</param>
/// <param name="facea">面方位角[deg</param>
public WallFaceConnector(string name, ISpaceBlock
space, ISurface surface, int surfacedirection, int
layerIndex,
 double area, double alphac, double alphar, double
 solarabsorptionrate, double longwaveemissionrate,
 double faceh, double facea)
{
 this.Name = name;
 this.Area = area;
 this.AlphaC = alphac;
 this.AlphaR = alphar;
 this.SolarAbsorptionRate = solarabsorptionrate;
 this.LongWaveEmissionRate = longwaveemissionrate;
 this.MoistureTransferRate = alphac * 0.01;

 this.Space = space;
 this.Surface = surface;
 this.SurfaceDirection = surfacedirection;
 this.layerIndex = layerIndex;

 this.FaceSinH = Math.Sin(faceh / 180.0 * Math.PI);
 this.FaceCosH = Math.Cos(faceh / 180.0 * Math.PI);
 this.FaceSinA = Math.Sin(facea / 180.0 * Math.PI);
 this.FaceCosA = Math.Cos(facea / 180.0 * Math.PI);
 this.FaceWz = this.FaceCosH;
 this.FaceWw = this.FaceSinH * this.FaceSinA;
 this.FaceWs = this.FaceSinH * this.FaceCosA;
}

public void Init(SimulationContext context)
{

}

/// <summary>
/// 初期化 (反復前に毎回))
/// </summary>
/// <param name="context"></param>
public void IterationInit(SimulationContext context)
{
 //
 //太陽とのなす角,表面の受ける日射量等を取得する
 //

 if (Space is OutsideBlock)
 {
  double sd = 0.0;
  double ss = 0.0;
  double sr = 0.0;
  double lw = 0.0;

  CosTheta = Space.GetCosTheta(FaceWz, FaceWw,
  FaceWs);
  Space.GetSolar(CosTheta, FaceCosH, out sd, out ss,
  out sr, out lw);
```

```
      DirectSolar = sd;
      SkySolar = ss;
      ReflectSolar = sr;
      LongWave = lw;
    }
  }

  public void Run(SimulationContext context)
  {

  }

  public void Commit(SimulationContext context)
  {

  }

  /// <summary>
  /// 対流熱流量[W/m2] スペースから面に向かう場合を正
  /// </summary>
  public virtual void AddHeat(double qc)    ///
  {
    Qc = qc;
  }

  /// <summary>
  /// 対流熱流量[W/m2] スペースから面に向かう場合を正
  /// </summary>
  public virtual double GetHeat()
  {
    return Qc;
  }

  /// <summary>
  /// 水分移動
  /// </summary>
  /// <param name="dJ">水分伝達量 (単位[kg]) </param>
  public virtual void AddMoisture(double dJ, IConnector
  src, IConnector kernel)
  {
    dJ_tick += dJ;
    System.Diagnostics.Debug.Assert(dJ != -0.0002);

    System.Diagnostics.Debug.WriteLine("AddHeat: [{0}]-
    >[{1}] {2}[kg] by {3}", src.Name, this.Name, dJ,
    kernel);
  }

  public virtual void AddMoisture(double dJ)
  {
    dJ_tick += dJ;
  }

  public virtual double GetMoisture()
  {
    return dJ_tick;
  }
 }
}
```

1.14 OutsideBlock.cs

```
using HLFX.Utility;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace HLFX.Block
{
 /// <summary>
 /// 外部環境
 /// </summary>
 public class OutsideBlock: BaseBlock, ISpaceBlock
 {
  /// <summary>
  /// 外気温度
  /// </summary>
  public double Temp { get; set; }

  /// <summary>
  /// 絶対湿度[kg/kg']
  /// </summary>
  public double Humidity { get; set; }

  /// <summary>
  /// 法線面直達日射量[W/m2]
  /// </summary>
  public double DSolar { get; set; }

  /// <summary>
  /// 水平面天空日射量[W/m2]
  /// </summary>
  public double SSolar { get; set; }

  /// <summary>
  /// 夜間放射量[W/m2]
  /// </summary>
  public double Lradiation { get; set; }

  private double SinH;
  private double CosH;
  private double SinA;
  private double CosA;

  /// <summary>
  /// 反射日射量(下向き水平面仮定)[W/m2]
  /// </summary>
  private double RSolar;

  /// <summary>
  /// ダミー
  /// </summary>
  public double MRT { get; set; }
  public double DirectSolarRadiationTotal { get; set; }
  public double DirectSolarConvectionTotal { get; set; }

  public double SkySolarRadiationTotal { get; set; }
  public double SkySolarConvectionTotal { get; set; }
  public double ReflectSolarRadiationTotal { get; set; }
  public double ReflectSolarConvectionTotal { get;
  set; }

  public double SensibleHeatGenerationRadiationToWall
  { get { return 0.0; } }

  public double SolarRadiationToFloor { get { return
  0.0; } }
```

```
public double SolarRadiationToWall { get { return
0.0; } }

/// <summary>
/// ファイル名
/// </summary>
public string FileName { get; set; }

/// <summary>
/// 気象データ
/// </summary>
private IList<SMASHWeatherData> dataSet;

public OutsideBlock()
{
 Name = "外部環境";
}

private void Update(DateTime dt)
{
 //補間値の取得
 var data =
 SMASHWeatherLoader.GetLinearInterpolated(dt.Month,
 dt.Day, dt.Hour, dt.Minute, dataSet);

 //設定
 Temp = data.Temperature;
 Humidity = data.Humidity;
 DSolar = data.Ids;
 SSolar = data.Iis;
 Lradiation = data.Ins;
 SinH = data.SinH;
 SinA = data.SinA;
 CosH = data.CosH;
 CosA = data.CosA;

 RSolar = 0.1 * (DSolar * SinH + SSolar);
}

/// <summary>
/// 面の角度を受け取り,太陽方向となす角のcosを返す
/// </summary>
/// <param name="facewz">面のcos(傾斜角)</param>
/// <param name="faceww">面のsin(傾斜角)×sin(方位
角)</param>
/// <param name="faceww">面のsin(傾斜角)×cos(方位
角)</param>
public double GetCosTheta(double facewz, double
faceww, double facews)
{
 return this.SinH * facewz + this.CosH * this.SinA *
 faceww + this.CosH * this.CosA * facews;
}

/// <summary>
/// 壁の場合のSAT温度を返す
/// </summary>
/// <param name="solarabsorptionrate">表面の日射吸収率
</param>
/// <param name="longwaveemissionrate">表面の長波放射
率</param>
/// <param name="alphat">表面のαt</param>
/// <param name="costheta">面と太陽方向のcosθ</param>
/// <param name="facecosh">面のcosH</param>
public double GetSAT(double solarabsorptionrate,
double longwaveemissionrate,double alphat, double
costheta, double facecosh)
{
 double a = 0.0;
 double sf = 0.5 * (1.0 + facecosh);    ///天空形態係
 数

 if (costheta > 0.0)
  a += solarabsorptionrate * DSolar * costheta;

 a += solarabsorptionrate * (sf * SSolar + (1.0 - sf)
 * RSolar);

 a -= longwaveemissionrate * sf * Lradiation;

 if (alphat > 0.0)
  a = a / alphat + Temp;
 else
  a = Temp;

 return a;
}

/// <summary>
/// 窓の場合のSAT温度を返す
/// </summary>
/// <param name="longwaveemissionrate">表面の長波放射
率</param>
/// <param name="alphat">表面のαt</param>
/// <param name="costheta">面と太陽方向のcosθ</param>
/// <param name="facecosh">面のcosH</param>
public double GetSAT(double longwaveemissionrate,
double alphat, double costheta, double facecosh)
{
 double a = 0.0;
 double sf = 0.5 * (1.0 + facecosh);    ///天空形態係
 数

 a -= longwaveemissionrate * sf * Lradiation;

 if (alphat > 0.0)
  a = a / alphat + Temp;
 else
  a = Temp;

 return a;
}

/// <summary>
/// 面の角度を受け取り,当該面に入射する日射量,夜間放射
量を返す
/// </summary>
public void GetSolar(double costheta, double facecosh,
out double directsolar, out double skysolar, out
double reflectsolar, out double longwave)
{
 double sf = 0.5 * (1.0 + facecosh);   ///面からみた天
 空形態係数

 if (costheta > 0.0)
  directsolar = DSolar * costheta;
 else
  directsolar = 0.0;

 skysolar = sf * SSolar;
 reflectsolar =  (1.0 - sf) * RSolar;
 longwave = sf * Lradiation;
}
```

```
    public override void Init(SimulationContext context)
    {
      //気象データ読込
      dataSet = SMASHWeatherLoader.ReadAllData(FileName);

      Update(context.Current);

      base.Init(context);
    }

    public override void IterationInit(SimulationContext
    context)
    {
      Update(context.Current);
    }

    public override void Commit(SimulationContext context)
    {
      ///Update(context.Current);

      base.Commit(context);
    }

    public  void SetFaceConnector(WallFaceConnector
    connectingface, int facetype)
    {
      ///何もしない
    }

    public void
    SetVentilation(VentilationHeatTransferBlock
    ventilation)
    {
      ///何もしない
    }

    public void AddVent(double qv, double jv)
    {
      ///何もしない
    }

  }
}
```

1.15 FixedConditionroomBlock.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace HLFX.Block
{
  public class FixedConditionRoomBlock : RoomBlock,
  ISpaceBlock
  {
    public struct ReferenceSpace
    {
      public double Ratio;       ///重み
      public ISpaceBlock Space;

      public ReferenceSpace(ISpaceBlock space, double
      ratio)
      {
        Space = space;
        Ratio = ratio;
      }

    }

    /// <summary>
    /// 参照するスペースと比率
    /// </summary>
    private List<ReferenceSpace> ListReferenceRoom = new
    List<ReferenceSpace> ();

    public double RatioSum { get; private set; }

    public void SetReferencespace(ISpaceBlock
    referencespace, double ratio)
    {
      ReferenceSpace space = new
      ReferenceSpace(referencespace, ratio);
      ListReferenceRoom.Add(space);
      RatioSum += ratio;
    }

    /// <summary>
    /// コンストラクタ
    /// </summary>
    public FixedConditionRoomBlock(string name,
    IList<ISpaceBlock> referencespaces, IList<Double>
    ratios) : base()
    {
      base.Name = name;

      for (int i = 0; i < referencespaces.Count; i++)
      {
        SetReferencespace(referencespaces[i], ratios[i]);
      }
    }


    public FixedConditionRoomBlock() : base()
    {
      RatioSum = 0.0;
    }

    public override void Init(SimulationContext context)
    {
      Temp = 20.0;
      Humidity = 0.010;

      Qvent = 0.0;
      Jvent = 0.0;
      Qac = 0.0;
      Jac = 0.0;
      DirectSolarRadiationTotal = 0.0;
      DirectSolarConvectionTotal = 0.0;
      SkySolarRadiationTotal = 0.0;
      SkySolarConvectionTotal = 0.0;
      ReflectSolarRadiationTotal = 0.0;
      ReflectSolarConvectionTotal = 0.0;
      SensibleHeatGenerationTotalConvection = 0.0;
      SensibleHeatGenerationTotalRadiation = 0.0;
      VaporGenerationTotal = 0.0;
    }
```

```
    public override void IterationInit(SimulationContext
    context)
    {
      if (RatioSum > 0.0)
      {
       double tempsum = 0.0;
       double humsum = 0.0;
       for (int i = 0; i < ListReferenceRoom.Count; i++)
       {
        tempsum += ListReferenceRoom[i].Space.Temp *
        ListReferenceRoom[i].Ratio;
        humsum += ListReferenceRoom[i].Space.Humidity *
        ListReferenceRoom[i].Ratio;
       }

       Temp = tempsum / RatioSum;
       Humidity = humsum / RatioSum;
      }
    }

    public override void Run(SimulationContext context)
    {

    }

    public override void Commit(SimulationContext context)
    {

    }

  }
}
```

1.16 PersonBlock.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace HLFX.Block
{
  /// <summary>
  /// 室ごとの在室人数
  /// 1人につき1インスタンスではなく，人のいる室1つにつき1
  インスタンスとする
  /// 複数室には適用しない
  /// </summary>
  public class PersonBlock : BaseBlock
  {
    private const double EvpHeat = 2510.0;

    public RoomBlock Room { get; set; } = null;


    public bool InRoom { get; private set; }

    public string name { get; private set; }


    /// <summary>
    /// 1人あたり全熱[W/人]
    /// </summary>
    public double MaxTotalHeat { get; set; }

    /// <summary>
    /// 作用温度24℃のときの顕熱[W/人]
    /// </summary>
    public double StdSensibleHeat { get; set; }

    /// <summary>
    /// 顕熱勾配(24℃との差に乗じる係数)
    /// </summary>
    public double SensibleHeatSlope { get; set; }

    /// <summary>
    /// 対流成分比率[-]
    /// </summary>
    public double ConvectionRate { get; set; }

    /// <summary>
    /// スケジュール
    /// </summary>
    public ScheduleSeries[,] PersonSchedule;

    /// <summary>
    /// コンストラクタ
    /// </summary>
    public PersonBlock(double maxtotalheat, double
    stdsensibleheat, double sensibleheatslope, double
    convectionrate,
     ScheduleSeries[,] schedule, RoomBlock room
     )
    {
     MaxTotalHeat = maxtotalheat;
     StdSensibleHeat = stdsensibleheat;
     SensibleHeatSlope = sensibleheatslope;
     ConvectionRate = convectionrate;

     PersonSchedule = new ScheduleSeries[3, 2];

     for (int term = 0; term < 3; term++)
     {
      for (int wd = 0; wd < 2; wd++)
      {
       PersonSchedule[term, wd] = schedule[term, wd];
      }
     }

     Room = room;
     room.SetPerson(this);
     this.Name = room.Name;
    }

    /// <summary>
    /// ある時点(期,曜日,時刻,分)を受け取り,顕熱発熱量・潜
    熱発熱量を返す
    /// <param name="time">時刻・分を4桁整数化したもの
    </param>
    /// <param name="qgenc">発熱量(対流成分)[W]</param>
    /// <param name="qgenr">発熱量(放射成分)[W]</param>
    /// <param name="jgen">水分量[kg/s]</param>
    /// </summary>
    public void GetHeat(SimulationContext context, out
    double qgenc, out double qgenr, out double jgen)
    {
     double scvalue = PersonSchedule[context.CurrentTerm,
     context.CurrentWeek].GetScheduleValue(context.Current
     Time);
```

```
  double ot = 0.5 * (Room.Temp + Room.MRT);
  double sensible = StdSensibleHeat - SensibleHeatSlope
  * (ot - 24.0);

  if (sensible > MaxTotalHeat) sensible = MaxTotalHeat;
  if (sensible < 0.0) sensible = 0.0;

  qgenc = sensible * ConvectionRate * scvalue;
  qgenr = sensible * (1.0 - ConvectionRate) * scvalue;
  jgen = (MaxTotalHeat - sensible) * scvalue / EvpHeat
  / 1000.0;    ///[kg/s]
}

/// <summary>
/// エアコンのON/OFFを返す
/// 0:OFF, 1:ON
/// <param name="time"></param> 時刻・分を4桁整数化し
たもの
/// <param name="roomtemp">室温[℃]</param>
/// <param name="outtemp">外気温[℃]</param>
/// <param name="windowopen">室の窓開け状態 0=閉じる,
1:開く</param>
/// <param name="aconoff">エアコンの運転状態 0=OFF,
1:ON</param>
/// </summary>
public void ACONOFF(SimulationContext context, double
roomtemp, double outtemp, int windowopen_before, int
aconoff_before,
  out int windowopen_new, out int aconoff_new)
{
  double scvalue_before =
  PersonSchedule[context.BeforeTerm,
  context.BeforeWeek].GetScheduleValue(context.BeforeTi
  me);
  double scvalue_current =
  PersonSchedule[context.CurrentTerm,
  context.CurrentWeek].GetScheduleValue(context.Current
  Time);

  if (scvalue_current == 0)      ///現時点が不在
  {
   windowopen_new = 0;
   aconoff_new = 0;
  }
  else
  {
   if (scvalue_before == 0)      ///前時点が不在(入室直
   後)
   {
    windowopen_new = 0;
    aconoff_new = 0;
   }
   else                          ///在室継続中
   {
    if (windowopen_before == 0 && aconoff_before == 0)
    ///CLOSE & OFF状態から
    {
     if (roomtemp < 29.5)
     {
      windowopen_new = 1;      ///窓開けに移行
      aconoff_new = 0;
     }
     else
     {
      windowopen_new = 0;
      aconoff_new = 1;       ///エアコンONに移行
     }
    }

    else if (windowopen_before == 1 && aconoff_before
    == 0)    ///OPEN & OFF状態から
    {
     if (roomtemp < 29.5 && 25.0 <= outtemp)
     {
      windowopen_new = 0;      ///CLOSE & OFFに戻る
      aconoff_new = 0;
     }
     else if (30.0 <= roomtemp)
     {
      windowopen_new = 0;      ///エアコンONに移行
      aconoff_new = 1;
     }
     else
     {
      windowopen_new = windowopen_before;      ///OPEN
      & OFFを維持
      aconoff_new = aconoff_before;
     }

    }

    else
    ///CLOSE & ON状態から
    {
     windowopen_new = windowopen_before;      ///CLOSE
     & ONを維持(退室するまで)
     aconoff_new = aconoff_before;
    }
   }
  }

 }

 public override void Commit(SimulationContext context)
 {
 }
}

}
```

1.17 AirConditionerBlock.cs

```
using static HLFX.Utility.Util;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace HLFX.Block
{
 public class AirConditionerBlock : BaseBlock
 {
  /// <summary>
  /// 空気比熱[J/kgK] 当面、定数とする
  /// </summary>
  private const double c_air = 1005;

  /// <summary>
```

```
/// 空気密度[kg/m3] 当面、定数とする
/// </summary>
private const double rho_air = 1.2;

/// <summary>
/// 蒸発潜熱[J/kg] 当面、定数とする
/// </summary>
private const double rL = 2510 * 1000;

/// <summary>
/// バイパスファクター[-]
/// </summary>
public double BypassFactor { get; private set; }

/// <summary>
/// 設定吹き出し風量[m^3/s]
/// </summary>
public double AirVolume { get; private set; }

/// <summary>
/// エアコンの冷房最大能力[kW]
/// </summary>
public double MaxCoolingCapacity { get; private set; }

/// <summary>
/// エアコンの動作タイプ
/// 0:スケジュールどおり, 1:スケジュールどおり&温度が
設定値未満ならOFF, 2:人体判定併用
/// </summary>
public int WorkType { get; private set; }

#region 状態

/// <summary>
/// 時刻Tの運転状態
/// </summary>
public bool currentOperation { get; set; }

/// <summary>
/// 時刻T+1の運転状態
/// </summary>
private bool nextOperation;


/// <summary>
/// 運転状態
/// </summary>
public bool Operation
{
 get
 {
  return currentOperation;
 }
 set
 {
  nextOperation = value;
 }
}

/// <summary>
/// 顕熱処理量[W] 冷房を正
/// </summary>
public double SensibleHeat { get; private set; }

/// <summary>
/// 潜熱処理量[kg/s] 冷房(除湿)を正 [W]に戻すにはrLを
乗じる
/// </summary>
public double LatentHeat { get; private set; }

/// <summary>
/// 吹き出し温度
/// </summary>
public double OutTemp { get; private set; }

/// <summary>
/// 吹き出し湿度[kg/kg']
/// </summary>
public double OutHumidity { get; private set; }


/// <summary>
/// スケジュール
/// </summary>
public ScheduleSeries[,] ACSchedule;

#endregion

#region 接続

/// <summary>
/// 居室
/// </summary>
public RoomBlock Room { get; private set; }

/// <summary>
/// 外部環境
/// </summary>
public OutsideBlock Outside { get; private set; }

#endregion

#region 計算結果

/// <summary>
/// 消費電力[kWh]
/// </summary>
public double E_AC { get; private set; }

/// <summary>
/// (計算過程)エアコンの理論効率[-]
/// </summary>
public double Efficiency { get; private set; }

/// <summary>
/// (計算過程)圧縮効率理論効率[-]
/// </summary>
public double EtaComp { get; private set; }

/// <summary>
/// (計算過程)COP[-]
/// </summary>
public double COP
{
 get
 {
  return Efficiency * EtaComp;
 }
}

#endregion

/// <summary>
/// コンストラクタ
```

```
/// </summary>
/// <param name="name">名称</param>
/// <param name="room">居室</param>
/// <param name="outside">外部環境</param>
public AirConditionerBlock(string name, RoomBlock
room, OutsideBlock outside,
 double maxheatingcapacity, double maxcoolingcapacity,
 double airvolume, double bypassfactor,
 ScheduleSeries[,] schedule)
{
 base.Name = name;
 this.Room = room;
 this.Outside = outside;
 this.MaxCoolingCapacity = maxcoolingcapacity;
 this.AirVolume = airvolume;
 this.BypassFactor = bypassfactor;

 ACSchedule = new ScheduleSeries[3, 2];

 for (int term = 0; term < 3; term++)
 {
  for (int wd = 0; wd < 2; wd++)
  {
   ACSchedule[term, wd] = schedule[term, wd];
  }
 }

 room.SetAirConditioner(this);
}

/// <summary>
/// 運転状態をセットする
/// 引数intのみ:人体判定結果を受け取る
/// </summary>
public void SetOperation(int aconoff)
{
 if (aconoff == 1)
  currentOperation = true;
 else
  currentOperation = false;
}

/// <summary>
/// 運転状態をセットする
/// 引数contextのみ:スケジュールのみに依存
/// </summary>
public void SetOperation(SimulationContext context)
{
 double scvalue = ACSchedule[context.CurrentTerm,
 context.CurrentWeek].GetScheduleValue(context.Current
 Time);

 if (scvalue > 0.0)
  currentOperation = true;
 else
  currentOperation = false;
}

/// <summary>
/// 運転状態をセットする
/// 引数contextと室温
/// </summary>
public void SetOperation(SimulationContext context,
double roomtemp)
{
 double scvalue = ACSchedule[context.CurrentTerm,
 context.CurrentWeek].GetScheduleValue(context.Current
 Time);
 double reftemp = 0.0;          ///ONOFF判定の参照温度
 (WorkType=0ならこのまま使う)

 if (WorkType == 1) reftemp = roomtemp;

 if (scvalue >= reftemp)
  currentOperation = true;
 else
  currentOperation = false;
}

public override void Run(SimulationContext context)
{
 ///
 ///エアコンONの冷房期のみ動作
 ///
 if (currentOperation == true && context.CurrentTerm
 == 1)
 {

  #region 1.前提条件

  double roomtemp = Room.Temp;
  ///室空気温度
  double roomabstemp =
  ToAbsoluteTemperature(roomtemp);    ///室空気絶対温
  度
  double roomah = Room.Humidity;
  ///室空気絶対湿度[kg/kg']
  double roomrh = CalcHumidity(roomtemp, roomah, 2);
  ///室空気相対湿度[-]

  #endregion


  double Ls = 0.0;    //顕熱処理量[W] テーブルから補
  間
  double Ll = 0.0;    //潜熱処理量[W] テーブルから補
  間

  GetProcessLoad(roomtemp, roomrh, out Ls, out Ll);

  SensibleHeat = Ls;
  LatentHeat = Ll / rL;   ///[W]を[kg/s]に換算

  #endregion

  #region 3.1 吹出し温湿度の算出

  OutTemp = roomtemp - Ls / (c_air * rho_air *
  AirVolume);
  OutHumidity = roomah - Ll / (rL * rho_air *
  AirVolume);

  #endregion

  #region 3.2 蒸発温度の算出

  //エアコンの蒸発温度 T_evp[℃]
  double evptemp = (OutTemp - roomtemp *
  BypassFactor) / (1.0 - BypassFactor); ///[℃]
  double absevptemp = ToAbsoluteTemperature(evptemp);
  ///[K]

  #endregion
```

```
#region 3.3 凝縮温度の算出

double cndtemp = Outside.Temp;

#endregion

#region 3.4 理論効率の算出
// ***h_cmp_in***
double temp_cmp_in = evptemp;                    //圧縮
機入口温度 T_cmp_in[℃]
double pres_evp = f_P_sgas(evptemp);             //蒸発
圧力 P_evp [MPa]
double pres_cmp_in = pres_evp;                   //圧縮
機入口圧力 P_cmp_in [MPa]
double h_cmp_in = f_H_gas_comp_in(pres_cmp_in,
temp_cmp_in); //圧縮機入口エンタルピー
h_cmp_in[kJ/kg]

// ***h_cmp_out***
double S_cmp_in = f_S_gas(pres_cmp_in, h_cmp_in);
//圧縮機入口エントロピー S_cmp_in [kJ/kgK]
double S_cmp_out = S_cmp_in;
//圧縮機出口エントロピー S_cmp_out [kJ/kgK]
double pres_cnd = f_P_sgas(cndtemp);
//凝縮圧力 P_cnd [MPa]
double P_cmp_out = pres_cnd;
//圧縮機出口圧力 P_cmp_out [MPa]
double h_cmp_out = f_H_gas_comp_out(P_cmp_out,
S_cmp_out); //圧縮機出口エンタルピー h_cmp_out
[kJ/kg]

// ***h_evp_in***
double temp_cnd_out = cndtemp;
//凝縮器出口温度 T_cnd_out [℃]
double h_cnd_out = f_H_liq(pres_cnd, temp_cnd_out);
//凝縮器出口エンタルピー h_cnd_out [kJ/kg]
double h_evp_in = h_cnd_out;                     //蒸
発器入口エンタルピー h_evp_in [kJ/kg]

// ***h_evp_out***
double h_evp_out = h_cmp_in;                     //蒸
発器出口エンタルピー h_evp_out [kJ/kg]

//エアコンの理論効率 e_AC
double e_AC = 0.0;

if (h_cmp_out - h_cmp_in > 0.0)
 e_AC = Math.Min((h_evp_out - h_evp_in) /
 (h_cmp_out - h_cmp_in), 10.0);
else
 e_AC = 10.0;

this.Efficiency = e_AC;

#endregion

#region 3.5 消費電力の算出

//***圧縮機の消費電力E_cmp [kW]***
double qc = (Ls + Ll) / 1000.0;
//処理能力[kW]
double eta = -0.1255 * qc * qc + 0.6902 * qc +
0.0104;  //圧縮機の圧縮効率[-]

if (eta <= 0.1)
 eta = 0.1;

  this.EtaComp = eta;

  double E_cmp = qc / (eta * e_AC);
  //圧縮機の消費電力E_cmp[kW]

  //***補機の消費電力E_aux[kW]***
  double E_aux = 0.0451;

  //エアコンの消費電力 E_AC [kWh]
  E_AC = (E_cmp + E_aux) * context.Dt / 3600.0;

  #endregion

  //熱・水分の移動
  Room.AddAC(SensibleHeat, LatentHeat);
 }
 else
  return;
}

private const double _dT_RM = 3.0;
private const double _dh_RM = 0.10;
private static readonly double[] _T_RM = new double[]
{ 24, 27, 30, 33, 9999 };
private static readonly double[] _h_RM = new double[]
{ 0.5, 0.6, 0.7, 0.8, 9999 };

/// <summary>
/// テーブル
/// </summary>
private static readonly double[,] _Ls = new double[,]
{
 {300, 1011, 1943, 1864, 9999},
 {300,  848, 1658, 1569, 9999},
 {300,  688, 1399, 1304, 9999},
 {300,  524, 1162, 1062, 9999},
 {9999, 9999,9999, 9999, 9999},
};

/// <summary>
///テーブル
/// </summary>
private static readonly double[,] _Ll = new double[,]
{
 {   0,    0, 1357, 1436, 9999},
 {   0,    0, 1642, 1731, 9999},
 {   0,  154, 1901, 1996, 9999},
 {   0,  349, 2138, 2238, 9999},
 {9999, 9999, 9999, 9999, 9999},
};

/// <summary>
/// 顕熱・潜熱処理量の算出
/// </summary>
/// <param name="T_RM">吸込み空気温度[℃]</param>
/// <param name="h_RM">吸込み空気相対湿度[-]</param>
/// <param name="Ls">顕熱処理量[W]</param>
/// <param name="Ll">潜熱処理量[W]</param>
public void GetProcessLoad(double roomtemp, double
roomrh, out double Ls, out double Ll)
{
 double T_RM = Math.Max(_T_RM[0], Math.Min(roomtemp,
 _T_RM[3]));  ///roomtempが_T_RMの範囲内ならroomtemp,
 範囲外なら適用する_T_RM
 double h_RM = Math.Max(_h_RM[0], Math.Min(roomrh,
 _h_RM[3]));    ///roomrhが_h_RMの範囲内ならroomrh, 範
 囲外なら適用する_h_RM
```

```
  int j = Array.FindIndex(_T_RM, x => T_RM < x) - 1;
  int i = Array.FindIndex(_h_RM, x => h_RM < x) - 1;

  double Ls_low = (_Ls[i, j + 1] - _Ls[i, j]) * (T_RM -
  _T_RM[j]) / _dT_RM + _Ls[i, j];
  double Ls_high = (_Ls[i + 1, j + 1] - _Ls[i + 1, j])
  * (T_RM - _T_RM[j]) / _dT_RM + _Ls[i + 1, j];
  Ls = (Ls_high - Ls_low) * (h_RM - _h_RM[i]) / _dh_RM
  + Ls_low;

  double Ll_low = (_Ll[i, j + 1] - _Ll[i, j]) * (T_RM -
  _T_RM[j]) / _dT_RM + _Ll[i, j];
  double Ll_high = (_Ll[i + 1, j + 1] - _Ll[i + 1, j])
  * (T_RM - _T_RM[j]) / _dT_RM + _Ll[i + 1, j];
  Ll = (Ll_high - Ll_low) * (h_RM - _h_RM[i]) / _dh_RM
  + Ll_low;

  //
  //調整が必要(もともと相対湿度50%未満の場合)
  //
  //顕熱のみ処理した温度で、相対湿度が90%未満の場合は潜
  熱処理しない
  //95%超の場合は90%まで下げる潜熱処理とする
  //

  if (roomrh < 0.5)
  {
    double temp = roomtemp - Ls / (c_air * rho_air *
    AirVolume);
    double abshumidity =
    Utility.Util.CalcHumidity(temp, roomrh, 1);

    double rh = Utility.Util.CalcHumidity(temp,
    abshumidity, 2);    ///顕熱だけ処理した場合の相対湿
    度

    if (rh < 0.9)
      Ll = 0.0;         ///相対湿度90%未満の場合は潜熱処
      理せず吹き出す
    else
    {
      double newabshumidity =
      Utility.Util.CalcHumidity(temp, 0.9, 1);  //相対湿
      度90%となる絶対湿度
      Ll = rL * rho_air * AirVolume * (abshumidity -
      newabshumidity);
    }
  }
}

/// <summary>
/// 付録A.7.1 式(13)
/// </summary>
/// <param name="T">冷媒の温度[℃]</param>
/// <returns>飽和蒸気の圧力[MPa]</returns>
private double f_P_sgas(double T)
{
  return
     2.75857926950901 * Math.Pow(10, -17) * Math.Pow(T,
     8)
    + 1.49382057911753 * Math.Pow(10, -15) *
    Math.Pow(T, 7)
    + 6.52001687267015 * Math.Pow(10, -14) *
    Math.Pow(T, 6)
    + 9.14153034999975 * Math.Pow(10, -12) *
    Math.Pow(T, 5)
    + 3.18314616500361 * Math.Pow(10, -9) * Math.Pow(T,
    4)
    + 1.60703566663019 * Math.Pow(10, -6) * Math.Pow(T,
    3)
    + 3.06278984019513 * Math.Pow(10, -4) * Math.Pow(T,
    2)
    + 2.54461992992037 * Math.Pow(10, -2) * Math.Pow(T,
    1)
    + 7.98086455154775 * Math.Pow(10, -1)
    ;
}

/// <summary>
/// 付録A.8.2 式(14)
/// </summary>
/// <param name="P">過熱蒸気の圧力[MPa]</param>
/// <param name="T">過熱蒸気の温度[℃]</param>
/// <returns>過熱蒸気の比エンタルピー[kJ/kg]</returns>
private double f_H_gas_comp_in(double P, double T)
{
  var K = T + 273.15;
  var K2 = K * K;
  var K3 = K2 * K;

  var P2 = P * P;
  var P3 = P2 * P;
  var P4 = P2 * P2;

  return
     -1.00110355 * Math.Pow(10, -1) * P3
     - 1.184450639 * Math.Pow(10, 1) * P2
     - 2.052740252 * Math.Pow(10, 2) * P
     + 3.20391 * Math.Pow(10, -6) * K3
     - 2.24685 * Math.Pow(10, -3) * K2
     + 1.279436909 * K
     + 3.1271238 * Math.Pow(10, -2) * P2 * K
     - 1.415359 * Math.Pow(10, -3) * P * K2
     + 1.05553912 * P * K
     + 1.949505039 * Math.Pow(10, 2)
    ;
}


/// <summary>
/// 付録A.8.2 式(15)
/// </summary>
/// <param name="P">過熱蒸気の圧力[MPa]</param>
/// <param name="T">過熱蒸気の比エントロピー[kJ/(kg・
K)]</param>
/// <returns>過熱蒸気の比エンタルピー[kJ/kg]</returns>
private double f_H_gas_comp_out(double P, double S)
{
  var P2 = P * P;
  var P3 = P2 * P;
  var P4 = P2 * P2;

  var S2 = S * S;
  var S3 = S2 * S;
  var S4 = S2 * S2;

  return
    -1.869892835947070 * 0.1 * P4
    + 8.223224182177200 * 0.1 * P3
    + 4.124595239531860 * P2
    - 8.346302788803210 * 10 * P
    - 1.016388214044490 * 100 * S4
    + 8.652428629143880 * 100 * S3
```

```
    - 2.574830800631310 * 1000 * S2
    + 3.462049327009730 * 1000 * S
    + 9.209837906396910 * 0.1 * P3 * S
    - 5.163305566700450 * 0.1 * P2 * S2
    + 4.076727767130210 * P * S3
    - 8.967168786520070 * P2 * S
    - 2.062021416757910 * 10 * P * S2
    + 9.510257675728610 * 10 * P * S
    - 1.476914346214130 * 1000
    ;
}

/// <summary>
/// 付録A.8.2 式(16)
/// </summary>
/// <param name="P">過熱蒸気の圧力[MPa]</param>
/// <param name="h">過熱蒸気の比エンタルピー
[kJ/kg]</param>
/// <returns>過熱蒸気のエントロピー
[kJ/(kg*K)]</returns>
private double f_S_gas(double P, double h)
{
  var P2 = P * P;
  var P3 = P2 * P;
  var P4 = P2 * P2;

  var h2 = h * h;
  var h3 = h2 * h;
  var h4 = h2 * h2;

  return
    5.823109493752840 * Math.Pow(10, -2) * P4
  - 3.309666523931270 * Math.Pow(10, -1) * P3
  + 7.700179914440890 * Math.Pow(10, -1) * P2
  - 1.311726004718660 * P
  + 1.521486605815750 * Math.Pow(10, -9) * h4
  - 2.703698863404160 * Math.Pow(10, -6) * h3
  + 1.793443775071770 * Math.Pow(10, -3) * h2
  - 5.227303746767450 * Math.Pow(10, -1) * h
  + 1.100368875131490 * Math.Pow(10, -4) * P3 * h
  + 5.076769807083600 * Math.Pow(10, -7) * P2 * h2
  + 1.202580329499520 * Math.Pow(10, -8) * P * h3
  - 7.278049214744230 * Math.Pow(10, -4) * P2 * h
  - 1.449198550965620 * Math.Pow(10, -5) * P * h2
  + 5.716086851760640 * Math.Pow(10, -3) * P * h
  + 5.818448621582900 * 10
  ;
}

/// <summary>
/// 付録A.8.3 式(17)
/// </summary>
/// <param name="P">過冷却液の圧力[MPa]</param>
/// <param name="T">過冷却液の温度[℃]</param>
/// <returns>過冷却液の比エンタルピー[kJ/kg]</returns>
private double f_H_liq(double P, double T)
{
  var K = T + 273.15;
  var K2 = K * K;
  var K3 = K2 * K;

  var P2 = P * P;
  var P3 = P2 * P;

  return
    1.7902915 * Math.Pow(10, -2) * P3
  + 7.96830322 * Math.Pow(10, -1) * P2
  + 5.985874958 * Math.Pow(10, 1) * P
  + 0 * K3
  + 9.86677 * Math.Pow(10, -4) * K2
  + 9.8051677 * Math.Pow(10, -1) * K
  - 3.58645 * Math.Pow(10, -3) * P2 * K
  + 8.23122 * Math.Pow(10, -4) * P * K2
  - 4.42639115 * Math.Pow(10, -1) * P * K
  - 1.415490404 * Math.Pow(10, 2)
  ;
 }

 public override void Commit(SimulationContext context)
 {


  //温度,絶対湿度をリセット

  SensibleHeat = 0.0;
  LatentHeat = 0.0;
  OutTemp = 0.0;
  OutHumidity = 0.0;
  E_AC = 0.0;
  Efficiency = 0.0;
  EtaComp = 0.0;


  //T+1の状態設定
  currentOperation = nextOperation;

  base.Commit(context);
 }

 public override void Init(SimulationContext context)
 {
  ///base.Dt = context.AirConditionerDt;
 }
 }
}
```

1.18 ConvectionHeatTransferBlock.cs

```
using HLFX.Block;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace HLFX.Block
{
 /// <summary>
 /// 対流熱伝達カーネル
 /// </summary>
 public class ConvectionHeatTransferBlock : BaseBlock
 {
  /// <summary>
  /// 壁または窓表面
  /// </summary>
  ///private readonly ISurface Surface;

  /// <summary>
  /// 居室または外部
  /// </summary>
```

```
    ///private readonly ISpaceBlock Space;

    private readonly WallFaceConnector WallFace;

    public ConvectionHeatTransferBlock(WallFaceConnector
    wallface)
    {
      WallFace = wallface;
    }

    public override void Run(SimulationContext context)
    {

      //対流熱伝達率[W/m2K]
      //double a_Fc = F.ConvectionHeatTransferRate;
      double alphac = WallFace.AlphaC;

      //壁表面積[m2]
      //double area = WallFace.Area;

      //居室空気温度[℃]
      double spacetemp = WallFace.SpaceTemp;

      //壁面表面温度[℃]
      double surfacetemp = WallFace.SurfaceTemp;

      //対流熱伝達量（単位[W/m2]）（スペースから面に移動す
      る場合を正、逆なら負）。
      double qc = alphac * (spacetemp - surfacetemp);

      //熱移動
      WallFace.AddHeat(qc);

    }
  }
}
```

1.19 RadiationHeatTransferBlock.cs

```
using HLFX.Block;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace HLFX.Block
{
  /// <summary>
  /// 放射熱伝達ブロック
  /// </summary>
  public class RadiationHeatTransferBlock : BaseBlock
  {
    /// <summary>
    /// 壁面/窓 1
    /// </summary>
    private readonly ILayerBlock R1;

    /// <summary>
    /// 壁面/窓 2
    /// </summary>
    private readonly ILayerBlock R2;

    /// <summary>
    /// 居室の全表面積[m^2]
    /// </summary>
    private double A { get; set; }

    public RadiationHeatTransferBlock(ILayerBlock src,
    ILayerBlock dst, double roomSurfaceArea)
    {
      R1 = src;
      R2 = dst;
      A = roomSurfaceArea;
    }

    public override void Run(SimulationContext context)
    {
      //対流熱伝達率[W/m2K]（当面の定数）
      const double a_r12 = 5.0;

      //壁表面積[m2]
      double A_R1 = R1.Area;
      double A_R2 = R2.Area;

      //壁面表面温度[℃]
      double T1 = R1.SurfaceTemp;
      double T2 = R2.SurfaceTemp;

      //対流熱伝達量（単位[？？？]）（居室から壁面に移動す
      る場合を正、逆なら負）。
      double q_r1to2 = a_r12 * (A_R1 * A_R2 / A) * (T1 -
      T2);

    }
  }
}
```

1.20 TotalHeatTransferBlock.cs

```
using HLFX.Block;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace HLFX.Block
{
  /// <summary>
  /// 総合熱伝達カーネル
  /// </summary>
  public class TotalHeatTransferBlock : BaseBlock
  {

    private readonly WallFaceConnector WallFace;

    public TotalHeatTransferBlock(WallFaceConnector
    wallface)
    {
      WallFace = wallface;
      ///this.src = src;
      ///this.dst = dst;
    }

    public override void Run(SimulationContext context)
    {
      //外部
```

```
    //OutsideBlock O = src;

    //壁面/窓
    //ILayerBlock F = dst as ILayerBlock;

    //総合熱伝達率[W/m2K]
    double alphat = WallFace.AlphaT;

    //放射熱伝達率[W/m2K]
    //double a_Fr = F.RadiationHeatTransferRate;

    //壁表面積[m2]
    //double area = WallFace.Area;

    //相当外気温度[℃]
    double SAT = WallFace.SpaceTemp;

    //壁面表面温度[℃]
    double surfacetemp = WallFace.SurfaceTemp;

    //総合熱伝達量（単位[W/m2]）（スペースから壁面に移動
    する場合を正、逆なら負）。
    double qt = alphat * (SAT - surfacetemp);

    //熱移動
    WallFace.AddHeat(qt);
   }
  }
}
```

1.21 VentilationHeatTransferBlock.cs

```
using HLFX.Block;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace HLFX.Block
{
 /// <summary>
 /// 換気熱移動カーネル
 /// </summary>
 public class VentilationHeatTransferBlock : BaseBlock
 {

  /// <summary>
  /// 上流側スペース
  /// </summary>
  private readonly ISpaceBlock UpperSpace;

  /// <summary>
  /// 下流側スペース
  /// </summary>
  private readonly ISpaceBlock LowerSpace;

  /// <summary>
  /// 風量[m3/s]
  /// </summary>
  public double AirVolume { get; set; }

  //public double ExhaustRate { get; set; }

  /// <summary>
  /// スケジュール
  /// </summary>
  public ScheduleSeries[,] Schedule;


  /// <summary>
  /// c_air：空気比熱[J/kgK]。当面、定数とする。
  /// </summary>
  private const double c_air  = 1005;

  /// <summary>
  /// ρ_air：空気密度[kg/m3]。当面、定数とする。
  /// </summary>
  private const double rho_air = 1.2;

  public VentilationHeatTransferBlock(ISpaceBlock
  upperspace, ISpaceBlock lowerspace, double airvolume,
  ScheduleSeries[,] schedule)
  {
   this.UpperSpace = upperspace;
   this.LowerSpace = lowerspace;
   this.AirVolume = airvolume;

   Schedule = new ScheduleSeries[3, 2];

   for (int term = 0; term < 3; term++)
   {
    for (int wd = 0; wd < 2; wd++)
    {
     Schedule[term, wd] = schedule[term, wd];
    }
   }

   ///
   ///下流の室に登録
   ///
   if (lowerspace is RoomBlock)
   {
    lowerspace.SetVentilation(this);
   }
  }


  public override void Run(SimulationContext context)
  {
   if (LowerSpace is OutsideBlock == false)
   {

    double scvalue = Schedule[context.CurrentTerm,
    context.CurrentWeek].GetScheduleValue(context.Curre
    ntTime);

    double upperspacetemp = UpperSpace.Temp;
    double lowerspacetemp = LowerSpace.Temp;

    double qv = c_air * rho_air * AirVolume * scvalue *
    (upperspacetemp - lowerspacetemp);  ///単位[W]

    double upperspacehumidity = UpperSpace.Humidity;
    ///[kg/kg']
    double lowerspacehumidity = LowerSpace.Humidity;
    ///[kg/kg']

    double jv = rho_air * AirVolume * scvalue *
    (upperspacehumidity - lowerspacehumidity);  ///単位
    [kg/s]
```

```
      //熱・水分移動(下流側のみに追加)
      LowerSpace.AddVent(qv, jv);


    }
   }
  }
}
```

1.22 ConvectionMoistureTransferBlock.cs

```
using HLFX.Block;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace HLFX.Block
{
 /// <summary>
 /// 水分伝達カーネル
 /// </summary>
 public class ConvectionMoistureTransferBlock : BaseBlock
 {

  private readonly WallFaceConnector WallFace;

  public
  ConvectionMoistureTransferBlock(WallFaceConnector
  wallFace)
  {
   WallFace = wallFace;
  }

  public override void Run(SimulationContext context)
  {

   //水分伝達率[kg/m2s(kg/kg' )]
   double alphax = WallFace.MoistureTransferRate;

   //壁表面積[m2]
   double area = WallFace.Area;

   //居室絶対湿度[kg/kg']
   double spacehumidity = WallFace.SpaceHumidity;

   //壁面絶対湿度[kg/kg']
   double surfacehumidity = WallFace.SurfaceHumidity;

   //水分伝達量（単位[？？？]）（居室から壁面に移動する
   場合を正、逆なら負）。
   double J = alphax * area * (spacehumidity-
   surfacehumidity);

   //水分移動
   WallFace.AddMoisture(J);
  }

  public override void Init(SimulationContext context)
  {
   ///base.Dt = context.WallDt;
  }
 }
}
```

1.23 Material.cs

```
using HLFX.Block;
using System;
using System.Collections.Generic;
using System.Data;
using System.IO;
using System.Linq;
using System.Text;

namespace HLFX.Utility
{
 /// <summary>
 /// 材料
 /// </summary>
 public class Material
 {
  /// <summary>
  /// 名称
  /// </summary>
  public string Name { get; set; }

  /// <summary>
  /// 熱伝導率[W/mK]
  /// </summary>
  public double 熱伝導率 { get; set; }

  /// <summary>
  /// 空気層熱抵抗[m2K/W]（空気層のみ>0とする）
  /// </summary>
  public double 空気層熱抵抗 { get; set; }

  /// <summary>
  /// 容積比熱[J/m3K]
  /// </summary>
  public double 容積比熱 { get; set; }

  /// <summary>
  /// 密度[kg/m3]（湿気計算材料のみ>0とする）
  /// </summary>
  public double 密度 { get; set; }

 }

 /// <summary>
 /// 材料と厚さの組み合わせ(直列)
 /// </summary>
 public struct WallMaterialList
 {
  public Material Material;
  public double Thickness;          // 宮島変更
  Depth→Thickness
 }

 public struct WallSection
 {
  public string SecName;                          ///セク
  ション名
  public double SectionAreaRatio;                 ///面積
  比率
```

```
  public int MaterialNum;                          ///材料
  数
  public List<WallMaterialList> materiallist;    ///(材料
  と厚さ)×材料数

  public WallSection(string name, double ratio, int
  matnum) : this()
  {
    SecName = name;
    SectionAreaRatio = ratio;
    MaterialNum = matnum;
    this.materiallist = new List<WallMaterialList>();
  }
}

/// <summary>
/// 壁仕様(複数断面)
/// </summary>
///
public class WallSpec
{
  /// <summary>
  /// 仕様名
  /// </summary>
  public string Name { get; set; }

  /// <summary>
  /// 断面数
  /// </summary>
  public int SectionNum { get; set; }

  public List<WallSection> wallsection;

  public WallSpec()
  {
    //Name = name;
    //SectionNum = 0;
    this.wallsection = new List<WallSection>();
  }

}



/// <summary>
/// 窓仕様
/// </summary>
public class WindowProperty
{
  /// <summary>
  /// U値
  /// </summary>
  public double UValue { get; set; }

  /// <summary>
  /// ηR値
  /// </summary>
  public double EtaR { get; set; }

  /// <summary>
  /// ηC値
  /// </summary>
  public double EtaC { get; set; }

  /// <summary>
  /// ガラス厚さ
  /// </summary>
    public double Thickness { get; set; }
  }

}
```

1.24 HeatGenerator.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace HLFX.Block
{
  public class HeatGenerator
  {

    private const double EvpHeat = 2510.0;
    /// <summary>
    /// 名称
    /// </summary>
    public string Name { get; set; }

    /// <summary>
    /// 最大顕熱発熱[W]
    /// </summary>
    public double MaxSensibleHeat { get; set; }

    /// <summary>
    /// 最大潜熱発熱[kg/s]
    /// </summary>
    public double MaxVaporGeneration { get; set; }

    /// <summary>
    /// 対流成分比率[-]
    /// </summary>
    public double ConvectionRate { get; set; }

    /// <summary>
    /// スケジュール
    /// </summary>
    public ScheduleSeries[,] HeatGenSchedule;

    /// <summary>
    /// コンストラクタ
    /// </summary>
    public HeatGenerator(string name, double
    maxsensibleheat, double maxvapor, double
    convectionrate,
      ScheduleSeries[,] schedule, IList<RoomBlock>
      applyrooms
     )
    {
      ///string[,] ScheduleID = new string[3, 2];
      Name = name;
      MaxSensibleHeat = maxsensibleheat;
      MaxVaporGeneration = maxvapor;
      ConvectionRate = convectionrate;

      HeatGenSchedule = new ScheduleSeries[3, 2];

      for (int term = 0; term < 3; term++)
```

```
    {
     for (int wd = 0; wd < 2; wd++)
     {
      HeatGenSchedule[term, wd] = schedule[term, wd];
     }
    }

    for (int i = 0; i < applyrooms.Count; i++)
    {
     RoomBlock applyroom = applyrooms[i];
     applyroom.SetHeatGenerator(this);
    }
   }

   /// <summary>
   /// ある時点(期,曜日,時刻,分)を受け取り,顕熱発熱量・潜熱発熱量を返す
   /// <param name="time"></param> 時刻・分を4桁整数化したもの
   /// <param name="qgenc"></param> 発熱量(対流成分)[W]
   /// <param name="qgenr"></param> 発熱量(放射成分)[W]
   /// <param name="jgen"></param>  水分量[kg/s]
   /// </summary>
   public void GetHeat(SimulationContext context, out
   double qgenc, out double qgenr, out double jgen)
   {
    double scvalue = HeatGenSchedule[context.CurrentTerm,
    context.CurrentWeek].GetScheduleValue(context.Current
    Time);

    qgenr = this.MaxSensibleHeat * (1.0 - ConvectionRate)
    * scvalue;
    qgenc = this.MaxSensibleHeat * ConvectionRate *
    scvalue;
    jgen = this.MaxVaporGeneration * scvalue;    ///[kg/s]
   }

}

public struct ScheduleUnit
{
 /// <summary>
 /// スケジュールユニット(開始時刻と値の組み合わせ)
 /// </summary>

 public int StartTime;              ///開始時刻 12時30分
 =1230
 public double ScheduleValue;       ///値
             ///終了時刻はない(次の配列の開始時刻を見る)

}

public class ScheduleSeries
{
 List<ScheduleUnit> ScheduleSet;
 //public ScheduleUnit[] ScheduleSet { get; set; }

 /// <summary>
 /// コンストラクタ
 /// </summary>
 /// <param name="scheduleunits"></param>
 public ScheduleSeries()
 {
  ScheduleSet = new List<ScheduleUnit>();
 }

 /// <summary>
 /// スケジュールユニット(開始時刻と値のセット)を追加する
 /// </summary>
 /// <param name="scheduleunits"></param>
 public void UnitAdd(int starttime, double scvalue)
 {
  ScheduleSet.Add(new ScheduleUnit { StartTime =
  starttime, ScheduleValue = scvalue });
 }

 /// <summary>
 /// スケジュール値を返す
 /// </summary>
 /// <param name="scheduleunits"></param>
 public double GetScheduleValue(int time)
 {
  for (int i = 0; i < ScheduleSet.Count - 1; i++)
  {
   if (time < ScheduleSet[i + 1].StartTime)    ///time
   が次のセットの開始時刻より早くなったところで確定
    return ScheduleSet[i].ScheduleValue;
  }

  ///ここまでで見つからなかった場合,最後のセットが適用
  される
  int j = ScheduleSet.Count - 1;
  return ScheduleSet[j].ScheduleValue;
 }

 }
}
```

1.25 SMASHWeatherLoader.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;

namespace HLFX.Utility
{
 /// <summary>
 /// SMASH形式読込データ
 /// </summary>
 public struct SMASHWeatherData
 {
  /// <summary>
  /// 気温 [℃]
  /// </summary>
  public double Temperature;

  /// <summary>
  /// 絶対湿度 [kg/kg']
  /// </summary>
  public double Humidity;

  /// <summary>
  /// 法線面直達日射量 [W/m2]
  /// </summary>
  public double Ids;

  /// <summary>
  /// 水平面天空日射量 [W/m2]
  /// </summary>
  public double Iis;
```

```
  /// <summary>
  /// 水平面夜間放射量 [W/m2]
  /// </summary>
  public double Ins;

  /// <summary>
  /// Sin(太陽高度)
  /// </summary>
  public double SinH;

  /// <summary>
  /// Cos(太陽高度)
  /// </summary>
  public double CosH;

  /// <summary>
  /// Sin(太陽方位角)
  /// </summary>
  public double SinA;

  /// <summary>
  /// Cos(太陽方位角)
  /// </summary>
  public double CosA;

  /// 曜日は取得しない
}

public static class SMASHWeatherLoader
{
  /// <summary>
  /// 各月日数
  /// </summary>
  private static readonly int[] cMonth = new int[]{ 31,
  28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

  /// <summary>
  /// kcal/h → Wの換算係数
  /// </summary>
  public const double kcaltoW = 1.163;

  /// <summary>
  ///   月、日、時刻を行番号に変換します。
  ///   SMASH形式想定、時刻は1～24を指定
  /// </summary>
  /// <param name="m"></param>
  /// <param name="d"></param>
  /// <param name="h"></param>
  public static int GetIndex(int m, int d, int h)
  {
    int lDay = d - 1;
    for (int i = 0; i < m - 1; i++)
    {
      lDay += cMonth[i];
    }
    lDay *= 24;

    var ret = lDay + h - 1;    ///1時スタート

    if (ret == -1) return 8759;    //1月1日0時=>8759
    return ret;
  }

  /// <summary>
  /// ファイル読込
  /// </summary>
  /// <param name="fileName">ファイル名</param>
  public static IEnumerable<SMASHWeatherData>
  ReadData(string fileName)
  {
    using (StreamReader reader = new
    StreamReader(fileName, Encoding.ASCII))
    {
      int lLineCount = 1;
      string lLine = "";
      SMASHWeatherData data;

      //パース処理用補助関数
      Func<string, int, int, int> GetValue = (label, a,
      b) =>
      {
        int lValue;
        if (int.TryParse(lLine.Substring(a - 1, b - a +
        1).Trim(), out lValue) == false)
        {
          throw new Exception(string.Format("{0}行目：{1}
          のデータが数値に変換出来ません。", lLineCount,
          label));
        }
        return lValue;
      };

      //地点番号を無視
      reader.ReadLine();
      lLineCount++;

      //25行ごとに365日
      for (int d = 0; d < 365; d++)
      {
        //一行無視
        reader.ReadLine();
        lLineCount++;

        for (int h = 0; h < 24; h++, lLineCount++)
        {
          #region 一行読込

          lLine = reader.ReadLine();

          //気温
          //1～4文字目：気温。10で除すことで単位[℃]とな
          る。
          data.Temperature = GetValue("気温", 1, 4) / 10.0;

          //絶対湿度
          //5～7文字目：絶対湿度。10,000で除すことで単位
          [kg/kg’]となる。
          data.Humidity = GetValue("絶対湿度", 5, 7) /
          10000.0;

          //法線面直達日射量
          //8～10文字目：法線面直達日射量。単位[kcal/m2h]
          data.Ids = GetValue("法線面直達日射量", 8, 10) *
          kcaltoW;

          //水平面天空日射量
          //11～13文字目：水平面天空日射量。単位[kcal/m2h]
          data.Iis = GetValue("水平面天空日射量", 11, 13) *
          kcaltoW;

          //水平面夜間放射量
          //14～16文字目：水平面夜間放射量。単位[kcal/m2h]
          data.Ins = GetValue("水平面夜間放射量", 14, 16) *
```

```
      kcaltoW;

      //太陽高度Hの正弦（sinH）
      //17～19文字目：太陽高度Hの正弦（sinH）。1,000で
      除す必要がある。
      data.SinH = GetValue("太陽高度Hの正弦（sinH）",
      17, 19) / 1000.0;

      //太陽高度Hの余弦（cosH）
      //20～22文字目：太陽高度Hの余弦（cosH）。1,000で
      除す必要がある。
      data.CosH = GetValue("太陽高度Hの余弦（cosH）",
      20, 22) / 1000.0;

      //太陽方位角Aの正弦（sinA）
      //23～26文字目：太陽方位角Aの正弦（sinA）。1,000
      で除す必要がある。太陽が真南より東側にある場合は
      数値の前に”－”が付く。
      data.SinA = GetValue("太陽方位角Aの正弦（sinA）",
      23, 26) / 1000.0;

      //太陽方位角Aの余弦（cosA）
      //27～30文字目：太陽方位角Aの余弦（cosA）。1,000
      で除す必要がある。太陽が真東・真西より北側にある
      場合は数値の前に”－”が付く。
      data.CosA = GetValue("太陽方位角Aの余弦（cosA）",
      27, 30) / 1000.0;

      #endregion

      yield return data;
    }
   }

   reader.Close();
  }
}

/// <summary>
/// ファイル読込
/// </summary>
/// <param name="fileName">ファイル名</param>
public static IList<SMASHWeatherData>
ReadAllData(string fileName)
{
 return ReadData(fileName).ToList();
}

/// <summary>
/// 分刻みの補間値を返します。
/// </summary>
/// <param name="m">月</param>
/// <param name="d">日</param>
/// <param name="h">時刻</param>
/// <param name="M">分</param>
public static SMASHWeatherData
GetLinearInterpolated(int m, int d, int h, int M,
IList<SMASHWeatherData> data)
{

 //経過時間
 int index_prev = GetIndex(m, d, h);    ///1月1日1時を0
 としたインデックス
 int index_next = index_prev + 1;       ///次の時刻のイ
 ンデックス

 //補間値
 if (index_prev == 8759 )               ///12月31日24時
 の次の時刻
 {
  index_next = 0;
 }
 double a = M / 60.0;
 var ret = new SMASHWeatherData
 {
  Temperature = (data[index_next].Temperature -
  data[index_prev].Temperature) * a +
  data[index_prev].Temperature,
  Humidity = (data[index_next].Humidity -
  data[index_prev].Humidity) * a +
  data[index_prev].Humidity,
  Ids = (data[index_next].Ids - data[index_prev].Ids)
  * a + data[index_prev].Ids,
  Iis = (data[index_next].Iis - data[index_prev].Iis)
  * a + data[index_prev].Iis,
  Ins = (data[index_next].Ins - data[index_prev].Ins)
  * a + data[index_prev].Ins,
  SinH = (data[index_next].SinH -
  data[index_prev].SinH) * a + data[index_prev].SinH,
  CosH = (data[index_next].CosH -
  data[index_prev].CosH) * a + data[index_prev].CosH,
  SinA = (data[index_next].SinA -
  data[index_prev].SinA) * a + data[index_prev].SinA,
  CosA = (data[index_next].CosA -
  data[index_prev].CosA) * a + data[index_prev].CosA,
 };

 //sin・cosの調整
 double b;
 b = ret.SinH * ret.SinH + ret.CosH * ret.CosH;
 if (b > 0.0)
 {
  ret.SinH /= Math.Sqrt(b);
  ret.CosH /= Math.Sqrt(b);
 }

 b = ret.SinA * ret.SinA + ret.CosA * ret.CosA;
 if (b > 0.0)
 {
  ret.SinA /= Math.Sqrt(b);
  ret.CosA /= Math.Sqrt(b);
 }

 return ret;
  }
 }
}
```

1.26 Util.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace HLFX.Utility
{
 public static class Util
 {
```

```
public const double AbsoluteTemperatureConstant =
273.16;

/// <summary>
/// 温度[℃]から絶対温度[K]への変換
/// </summary>
/// <param name="T">温度[℃]</param>
/// <returns>絶対温度[K]</returns>
public static double ToAbsoluteTemperature(double T)
{
 double Tk = T + AbsoluteTemperatureConstant;

 return Tk;
}

/// <summary>
/// 絶対温度[K]から温度[℃]への変換
/// </summary>
/// <param name="Tk">絶対温度[K]</param>
/// <returns>温度[℃]</returns>
public static double ToRelativeTemperature(double Tk)
{
 double T = Tk - AbsoluteTemperatureConstant;

 return T;
}

/// <summary>
/// 温度T[℃]の空気の飽和水蒸気圧[Pa]を求める。
/// </summary>
/// <param name="T">絶対温度[℃]</param>
/// <returns>温度T[℃]の空気の飽和水蒸気圧
[Pa]</returns>
public static double GetSaturatedVaporPressure(double
Tk)
{
 const double a1 = -6096.9385;
 const double a2 = 21.2409642;
 const double a3 = -2.711193 * 0.01;
 const double a4 = 1.673952 * 0.00001;
 const double a5 = 2.433502;

 double  Ps = Math.Exp(a1 / Tk + a2 + a3 * Tk + a4 *
 Tk * Tk + a5 * Math.Log(Tk));

 return Ps;
}

/// <summary>
/// 水蒸気圧[Pa]から絶対湿度[kg/kg']を求める。
/// </summary>
/// <param name="p">水蒸気圧[Pa]</param>
/// <returns>絶対湿度[kg/kg']</returns>
public static double GetAtmAbsolute(double p)
{
 const double Pa = 101325;   //大気圧

 double X = (0.622 * p) / (Pa - p);

 return X;
}

/// <summary>
/// 温度と絶対湿度から相対湿度を求める。
/// </summary>
/// <param name="Tk">絶対温度[K]</param>
/// <param name="X">絶対湿度[kg/kg']</param>
/// <returns>相対湿度[%]</returns>
public static double GetRelativeHumidity(double Tk,
double X)
{
 //温度Tから飽和絶対湿度Xsを計算
 double Ps = GetSaturatedVaporPressure(Tk);
 double Xs = GetAtmAbsolute(Ps);

 //飽和絶対湿度Xsと絶対湿度Xから相対湿度を求める
 double h = X / Xs * 100.0;

 return h;
}

/// <summary>
/// 温度と絶対湿度から相対湿度,または温度と相対湿度か
ら絶対湿度を求める。
/// Wexler-Hyland式
/// </summary>
/// <param name="temp">温度[℃]</param>
/// <param name="humidity">絶対湿度[kg/kg']または相対
湿度[-]</param>
/// <param name="calctype">1:絶湿計算, 2:相湿計算, 3:
相湿計算(100%超を認める)</param>
/// <returns>相対湿度[%]</returns>
public static double CalcHumidity(double temp, double
humidity, int calctype)
{
 const double C1 = -5800.2206;
 const double C2 = 1.3914993;
 const double C3 = -0.048640239;
 const double C4 = 0.000041764768;
 const double C5 = -0.000000014452093;
 const double C6 = 6.5459673;

 double abstemp = temp + AbsoluteTemperatureConstant;

 double matpress = C1 / abstemp + C2 + abstemp * (C3 +
 abstemp * (C4 + abstemp * C5))
    + C6 * Math.Log(abstemp);

 matpress = Math.Exp(matpress); ///[Pa]

 double matabshumidity = 0.62198 * matpress /
 (101325.0 - matpress); ///[kg/kg']

 if (calctype == 1)                    ///絶湿を計算
  return matabshumidity * humidity;     ///humidityは
  相対湿度

 else                                  ///相湿を計算
 {
  double rh = humidity / matabshumidity;
  ///humidityは絶対湿度

  if (calctype == 3)
   return rh;                    ///calctype=3のとき
   は1超でもそのまま返す
  else
  {
   if (rh > 1.0) rh = 1.0;
   return rh;
  }
 }
}
}
```