```cpp
O_METHOD_BEGIN(T1dComponent, InscribedCircle, "Inscribed Circle", 0, FLG_METHOD_SKIP_GUI_ARG, "Inscribed Circle")
{
  if (VARLID_ARG)
  {
    // 1.
    int error = object->updateInscribedCircleCalculator();
    if (error != 0)
      return error;
    // 2.
    if (!object->getInscribedCircleCalculator(false))
      return -1;
    // 3.
    w_QDialog dlg(core_Application::core());
    dlg.setWindowTitle("Inscribed Circle");
    QVBoxLayout* grid = new QVBoxLayout(&dlg);
    InscribedCircleCalculatorWidget* _InscribedCircleCalculatorWidget = new InscribedCircleCalculatorWidget();
    grid->addWidget(_InscribedCircleCalculatorWidget);
    // 4.
    _InscribedCircleCalculatorWidget->setInscribedCircleCalculator(object->getInscribedCircleCalculator());

    if (dlg.exec() == w_QDialog::Accepted)
    {

    }
    delete _InscribedCircleCalculatorWidget;
  }
}
O_METHOD_END;

InscribedCircleCalculator* T1dComponent::getInscribedCircleCalculator(bool CreatedAsNeeded)
{
  QString name = "InscribedCircleCalculator";
  InscribedCircleCalculator* _InscribedCircleCalculator = dynamic_cast<InscribedCircleCalculator*>(child(name));

  if (!_InscribedCircleCalculator && CreatedAsNeeded)
  {
    _InscribedCircleCalculator = dynamic_cast<InscribedCircleCalculator*>(TObject::new_object("InscribedCircleCalculator", name, this));
  }
  if (_InscribedCircleCalculator)
  {
    SET_OBJ_HIDE(_InscribedCircleCalculator);
    return _InscribedCircleCalculator;
  }
  return nullptr;
}
```

1dComponent.cpp

```cpp
int T1dComponent::setInscribedCircleCalculator()
{
  // pHubContour, pShroudContour
  int error = -1;

  // 1.
  QVector<TNurbsCurve*> TNCurves;
  // 2.
  if (pHubContour1)
    TNCurves.push_back(pHubContour1);
  if (pShroudContour)
    TNCurves.push_back(pShroudContour);
  else
    return error;
  // 3.
  InscribedCircleCalculator* InscribedCircleCalculator = getInscribedCircleCalculator();
  if (!InscribedCircleCalculator)
    return error;
  // 4.
  error = InscribedCircleCalculator->setTNCurve(TNCurves);

  return error;
}

int T1dComponent::updateInscribedCircleCalculator()
{
  int error = -1;
  // 1.
  error = setInscribedCircleCalculator();
  if (error != 0)
    return error;
  // 2.
  InscribedCircleCalculator* InscribedCircleCalculator = getInscribedCircleCalculator();
  // 3.
  error = InscribedCircleCalculator->createInscribedCircle();

  return error;
}
```

InscribedCircleCalculator.h /InscribedCircleCalculatorWidget.h

```h
class T_EXPORT_1D InscribedCircleCalculator : public TObject
{
  T_OBJECT;
  static const int n_tr = 10; // Number of tangent circles, member variables

private:
  double l_cc; // Length of CircleCenter line
  QVector<Double2> CircleCenters = QVector<Double2>(); // Center of all circles
  QVector<Double2> Points_hub = QVector<Double2>(); // all the points on the hub
  QVector<Double2> Points_shroud = QVector<Double2>(); // all the points on the shroud
  QVector<double> Qradius = QVector<double>(); // all the radius of the tangent circle
  QVector<Double2> L_Area = QVector<Double2>(); // all the points on the Areashow
public:
  InscribedCircleCalculator(QString object_n = "", TObject* iparent = NULL);
  virtual ~InscribedCircleCalculator();
  enum { ZR = 0, Area = 1, CurveTypeEnd };
private:
  QVector<curve_Circle*> getInscribedCircle(curve_Nurbs* c1, curve_Nurbs* c2, int num_tr = 10, doubl
  QVector<curve_Circle*> getInscribedCircle(curve_Nurbs& c1, curve_Nurbs& c2, int num_tr = 10, doubl
  int calculateInscribedCircle(curve_Nurbs* c1, curve_Nurbs* c2, int num_tr = n_tr, double tol = tol
public:
  QStringList getAllTypeNames();
  QString getTypeName(int type);
  curve_Topology* getTopo(int type);
  curve_Curve* getCurve(int type);
  QString getNurbsName(int type = 0, int CurveID = 0);
  curve_Nurbs* getNurbs(int type = 0, int CurveID = 0, bool createIfNotAvaiable = true);
  QString getCircleName(int type = 0, int CurveID = 0);
  curve_Circle* getCircle(int type = 0, int CurveID = 0, bool createIfNotAvaiable = true);
  curve_Nurbs* getCenterCurve(int type, bool createIfNotAvaiable = true);
  curve_Nurbs* getAreaCurve(int type, bool createIfNotAvaiable = true);
public:
  int setTNCurve(QVector<TNurbsCurve*> TNCurves);
  int setCNCurve(QVector<curve_Nurbs*> CNCurves);
  int setCircleCurve(QVector<curve_Circle*> ICCurves);
  int setCenterLine(QVector<curve_Circle*> ICCurves);
  int calculateArea();
  int createAreaLine();
public:
  int LoadCurves();
  int newLoadCurves();
  int newCalculateCrossSection();
public:
  int createInscribedCircle();
};
#endif
```

```h
#ifndef INSCRIBEDCIRCLECALCULATORWIDGET_H
#define INSCRIBEDCIRCLECALCULATORWIDGET_H

#include "w_TTWidget.h"
#include "InscribedCircleCalculator.h"

class w_PropertyHolderWidget;
class draw_TopologyInteractiveEditorWidget;
class w_QPushButton;

class T_EXPORT_1D InscribedCircleCalculatorWidget: public w_TTWidget
{
  Q_OBJECT;

public:
  InscribedCircleCalculatorWidget(QWidget* parent = 0);
  void setInscribedCircleCalculator(InscribedCircleCalculator* ICC);
  virtual ~InscribedCircleCalculatorWidget();

private:
  w_PropertyHolderWidget* holder;
  w_PropertyHolderWidget* holder_zrCurveWidget;
  w_PropertyHolderWidget* holder_AreaWidget;
  w_PropertyHolderWidget* holder_ConfiglWidget;

  draw_TopologyInteractiveEditorWidget* _zrCurveWidget;
  draw_TopologyInteractiveEditorWidget* _AreaWidget;

  w_QPushButton* btn_LoadCurves;
  w_QPushButton* btn_CalculateCrossSection;

  InscribedCircleCalculator* _InscribedCircleCalculator;

private:
  void update_zrCurveShow();
  void update_areaCurveShow();

private slots:
  void onLoadCurves();
  void onCalculateCrossSection();
};

#endif
```

InscribedCircleCalculatorWidget.cpp

```cpp
InscribedCircleCalculatorWidget::InscribedCircleCalculatorWidget(QWidget* parent) : w_TTWidget(parent)
{
  // W01
  QGridLayout* grid = new QGridLayout;
  // W02
  w_PropertyHolderWidget* holder = new w_PropertyHolderWidget();

  // W031 Add ZR sections topology show
  // a. w_PropertyHolderWidget* holder_zrCurveWidget;
  holder_zrCurveWidget = holder->getHolder(0, 0, 1, 2, tr("ZR Sections"));
  // b. draw_TopologyInteractiveEditorWidget* _zrCurveWidget;
  _zrCurveWidget = new draw_TopologyInteractiveEditorWidget(holder_zrCurveWidget);
  // c. placeWidget
  holder_zrCurveWidget->placeWidget(_zrCurveWidget);

  // W032 Add CrossSectionsArea Curve
  holder_AreaWidget = holder->getHolder(0, 2, 1, 2, tr("Cross Sections Area"));
  _AreaWidget = new draw_TopologyInteractiveEditorWidget(holder_AreaWidget);
  holder_AreaWidget->placeWidget(_AreaWidget);

  // W033 Button Config
  holder_ConfiglWidget = holder->getHolder(1, 0, 1, 4, tr("Config"));
  // a.
  btn_LoadCurves = holder_ConfiglWidget->addButton(QObject::tr("Load curves"), 0, 0, 1, 1);
  connect(btn_LoadCurves, SIGNAL(clicked()), this, SLOT(onLoadCurves()));
  // b.
  btn_CalculateCrossSection = holder_ConfiglWidget->addButton(QObject::tr("Calculate crossSection"), 0, 1, 1, 1);
  connect(btn_CalculateCrossSection, SIGNAL(clicked()), this, SLOT(onCalculateCrossSection()));

  // W04
  grid->addWidget(holder);
  // W05
  setLayout(grid);
  setFocusPolicy(Qt::StrongFocus);
}

void InscribedCircleCalculatorWidget::update_zrCurveShow()
{
  if (!_zrCurveWidget)
    return;
  if (_InscribedCircleCalculator->getTopo(0))
  {
    _zrCurveWidget->setSizeHint(QSize(600, 400));
    _zrCurveWidget->setTopology(_InscribedCircleCalculator->getTopo(0));
  }
}
```

InscribedCircleCalculatorWidget.cpp

```cpp
void InscribedCircleCalculatorWidget::setInscribedCircleCalculator(InscribedCircleCalculator* ICC)
{
  if (!ICC)
    return;
  else
    _InscribedCircleCalculator = ICC;

  update_zrCurveShow();
  update_areaCurveShow();
}

void InscribedCircleCalculatorWidget::onLoadCurves()
{
  if (!_zrCurveWidget)
    return;
  // W11
  //_InscribedCircleCalculator->newLoadCurves();
  _InscribedCircleCalculator->LoadCurves();
  // W12
  if (curve_Topology* topology = _InscribedCircleCalculator->getTopo(0))
  {
    _zrCurveWidget->setSizeHint(QSize(1000, 1000));
    // W13
    _zrCurveWidget->setTopology(topology);
  }
}

void InscribedCircleCalculatorWidget::onCalculateCrossSection()
{
  if (!_AreaWidget)
    return;
  // W21
  _InscribedCircleCalculator->newCalculateCrossSection();
  // W22
  if (curve_Topology* topology = _InscribedCircleCalculator->getTopo(1))
  {
    _AreaWidget->setSizeHint(QSize(1000, 600));
    // W23
    _AreaWidget->setTopology(topology);
  }
}

InscribedCircleCalculatorWidget::~InscribedCircleCalculatorWidget()
{
  if (_InscribedCircleCalculator) delete _InscribedCircleCalculator;
}
```

InscribedCircleCalculator.cpp

```cpp
REGISTER_OBJECT_CLASS(InscribedCircleCalculator, "Inscribed Circle Calculator", TObject);

InscribedCircleCalculator::InscribedCircleCalculator(QString object_n, TObject* iparent): TObject(object_n, iparent)
{
  INIT_OBJECT;
  l_cc = 0.;
}

// 0.Method 1
QVector<curve_Circle*> InscribedCircleCalculator::getInscribedCircle(curve_Nurbs* c1, curve_Nurbs* c2, int num_tr, double tol)
{
  // QVector<curve_Nurbs*> NCurves -> QVector<curve_Circle*> InscribedCircleCurves
  // 1.QVector
  QVector<curve_Circle*> ICCurves;
  QVector<Double2> QCenter;
  QVector<double> QRadius;

  // 2.getCurve
  curve_Nurbs* hub = c1;
  curve_Nurbs* shroud = c2;
  if (!hub || !shroud)
    return {};

  int num_radius = 10000; // Number of radius iterations
  double dus = 1.0 / (num_tr - 1);
  int num_uh = 1E7 * tol;
  double us = 0;
  double radius = 0.;

  // 3. getCenters and Radius
  for (int k = 0; k < num_tr; k++)
  {
    // 3.1 get Point
    Double2 point_A = shroud->getPoint(us);
    // 3.2 Tangential direction
    Double2 tangential_A = shroud->getTangent(us);
    // 3.3 mag
    double length = tangential_A.length();
    tangential_A /= length;
    // 3.4 normal
    Double2 normal_A = tangential_A.rotate(PI / 2.);
    // 3.5 radius
    double distance = (point_A - hub->getPoint(us)).length();
    radius = distance / 3;
```

```
for (int i = 0; i < num_radius; i++)
{
  // Number of intersections.
  int num_r = 0;
  double dl = 0.;
  double uh = 0.;
  // Circlecenter
  Double2 Circlecenter = (point_A - radius * normal_A);

  for (int j = 0; j ≤ num_uh; j++)
  {
    // get hub Point
    uh = 1. * j / num_uh;
    Double2 point_h = hub->getPoint(uh);
    // Distance between two points
    dl = (point_h - Circlecenter).length() - radius;
    if (dl < -tol)
      num_r += 1;
    if (num_r == 1)
      break;
    continue;
  }
  switch (num_r)
  {
  case 1:
    if (i == 0)
    {
      radius = distance / 10;
      continue;
    }
    QRadius.push_back(radius);
    break;
  case 0:
    radius *= 1. + 50 * tol;
    continue;
  }
  // eprintf("%d", i);
  break;
}

// 3.6 Circlecenter
Double2 Circlecenter = (point_A - radius * normal_A);
QCenter.push_back(Circlecenter);
```

InscribedCircleCalculator.cpp

```
// 4. getCircle
for (int i = 0; i < num_tr; i++)
{
  // 4.1
  curve_Circle* cc = new curve_Circle;
  // 4.2
  cc->setRadius(QRadius[i]);
  cc->setCenter(QCenter[i]);
  // 4.3
  ICCurves.push_back(cc);
}
return ICCurves;
}

int InscribedCircleCalculator::calculateInscribedCircle(curve_Nurbs* c1, curve_Nurbs* c2, int num_tr, double tol

QVector<curve_Circle*> InscribedCircleCalculator::getInscribedCircle(curve_Nurbs& c1, curve_Nurbs& c2, int num_t
{
  // QVector<curve_Nurbs*> NCurves -> QVector<curve_Circle*> InscribedCircleCurves
  // 1.QVector
  QVector<curve_Circle*> ICCurves;
  QVector<Double2> QCenter;
  QVector<double> QRadius;

  // 2.Init
  int num_radius = 10000; // Number of radius iterations
  double dus = 1.0 / (num_tr - 1);
  int num_uh = 1E7 * tol;
  double us = 0;
  double radius = 0.;

  // 3. getCenters and Radius
  for (int k = 0; k < num_tr; k++)
  {
    // 3.1 get Point
    Double2 point_A = c2.getPoint(us);
    // 3.2 Tangential direction
    Double2 tangential_A = c2.getTangent(us);
    // 3.3 mag
    double length = tangential_A.length();
    tangential_A /= length;
    // 3.4 normal
    Double2 normal_A = tangential_A.rotate(PI / 2.);
    // 3.5 radius
    double distance = (point_A - c1.getPoint(us)).length();
    radius = distance / 3;
```

```cpp
int InscribedCircleCalculator::setTNCurve(QVector<TNurbsCurve*> TNCurves)
{
  // TNurbsCurve -> Curve_Nurbs
  int error = -1;

  // 1.QVector
  QVector<curve_Nurbs*> CNCurves;
  for (int i = 0; i < TNCurves.size(); i++)
  {
    if (!TNCurves[i])
      return error;
    curve_Nurbs* c = new curve_Nurbs;

    // 2. transfer
    c->fillFromNurbsCurve(*TNCurves[i]);
    CNCurves.push_back(c);
  }

  error = setCNCurve(CNCurves);

  return error;
}

int InscribedCircleCalculator::setCNCurve(QVector<curve_Nurbs*> CNCurves)
{
  // NurbsCurve -> _ZRcurve
  int error = -1;
  if (CNCurves.size() < 2)
    return error;
  // 1
  for (int i = 0; i < CNCurves.size(); i++)
  {
    if (!CNCurves[i])
      return error;
    if (CNCurves[i]->getControlPointCount() < 2)
      return error;
  }

  // 2. get shroud/hub
  for (int i = 0; i < CNCurves.size(); i++)
  {
    if (curve_Nurbs* zrCurve = getNurbs(0, i))
      zrCurve->copyFrom(CNCurves[i]);
  }
  return 0;
}
```

```cpp
int InscribedCircleCalculator::setCircleCurve(QVector<curve_Circle*> ICCurves)
{
  int error = -1;
  for (int i = 0; i < n_tr; i++)
  {
    curve_Circle* circle = getCircle(0, i);
    if (!circle)
      return error;
    else
    {
      circle->setRadius(ICCurves[i]->getRadius());
      circle->setCenter(ICCurves[i]->getCenter());
    }
  }
  return 0;
}
```

```cpp
int InscribedCircleCalculator::setCenterLine(QVector<curve_Circle*> ICCurves)
{
  int error = -1;
  // 1.
  curve_Nurbs* ccc = getCenterCurve(0);
  QVector<Double2>Centers;
  // 2.
  for (int i = 0; i < n_tr; i++)
  {
    Centers.push_back(ICCurves[i]->getCenter());
  }
  if (!ccc)
    return error;
  else
    ccc->fitBezier(Centers);
  return 0;
}
```

```cpp
int InscribedCircleCalculator::LoadCurves()
{
  int error = -1;
  curve_Nurbs* hub = (getNurbs(0, 0));
  curve_Nurbs* shroud = (getNurbs(0, 1));
  if (!hub || !shroud)
    return error;

  QVector<curve_Circle*> ICCurves = getInscribedCircle(*hub, *shroud, n_tr);
  error = setCircleCurve(ICCurves);
  error = setCenterLine(ICCurves);
  return 0;
}

int InscribedCircleCalculator::newLoadCurves() { ... }

int InscribedCircleCalculator::newCalculateCrossSection()
{
  int error = -1;
  // 1.
  CircleCenters.clear();
  Points_hub.clear();
  Points_shroud.clear();
  Qradius.clear();
  L_Area.clear();
  // 2.
  curve_Nurbs* c1 = (getNurbs(0, 0));
  curve_Nurbs* c2 = (getNurbs(0, 1));
  if (!c1 || !c2)
    return error;
  // 3.
  error = calculateInscribedCircle(c1, c2);
  // 4.
  curve_Nurbs* ccc = getCenterCurve(0);
  ccc->fitBezier(CircleCenters);
  l_cc = ccc->getLength();
  // 5.
  error = calculateArea();
  error = createAreaLine();
  return 0;
}
```

```cpp
int InscribedCircleCalculator::calculateArea()
{
  int error = -1;
  if (CircleCenters.size() == 0)
    return error;

  for (int i = 0; i < n_tr; i++)
  {
    // 1. b
    Double2 Point_A = Points_shroud[i];
    Double2 Point_B = Points_hub[i];
    double s = (Point_A - Point_B).length(); // AB Chord length
    double p = Qradius[i]; // radius of the tangent circle
    double b = 2. / 3 * (s + p); // AEB Arc length

    // 2. Rc
    Double2 Point_D = 0.5 * (Point_A + Point_B);
    Double2 Point_C = Point_D + (1. / 3) * (CircleCenters[i] - Point_D);
    double Rc = Point_C[1]; // The radius of the axis of C

    // 3. L/F
    double L = 1. * i / (n_tr - 1) * l_cc; // i-Length of center line
    double F = 2. * PI * Rc * b; // The CrossSectionArea
    Double2 l_area_i = { L,F };
    L_Area.push_back(l_area_i);
  }
  return 0;
}

int InscribedCircleCalculator::createAreaLine()
{
  int error = -1;
  curve_Nurbs* ac = getAreaCurve(1);
  if (!ac)
    return error;
  else
    ac->fitBezier(L_Area);
  return 0;
}
```

InscribedCircleCalculator.cpp

```cpp
REGISTER_OBJECT_CLASS(InscribedCircleCalculator, "Inscribed Circle Calculator", TObject);

InscribedCircleCalculator::InscribedCircleCalculator(QString object_n, TObject* iparent): TObject(object_n, iparent)
{
  INIT_OBJECT;
  l_cc = 0.;
}

// 1.getTopo/Curve
QStringList InscribedCircleCalculator::getAllTypeNames()
{
  QStringList allTypeNames = QStringList() << "ZR" << "Area";
  return allTypeNames;
}

QString InscribedCircleCalculator::getTypeName(int type)
{
  if (type < ZR)
    type = ZR;
  if (type >= CurveTypeEnd)
    type = CurveTypeEnd - 1;
  return getAllTypeNames()[type];
}

curve_Topology* InscribedCircleCalculator::getTopo(int type)
{
  // 1.
  QString typeName = getTypeName(type);
  curve_Topology* T = dynamic_cast<curve_Topology*>(child(typeName));

  // 2.
  if (!T)
    T = (curve_Topology*)TObject::new_object("curve_Topology", typeName, this);
  if(T)
    return T;
  else
    return nullptr;
}
```

InscribedCircleCalculator.cpp

```cpp
curve_Curve* InscribedCircleCalculator::getCurve(int type)
{
  // 1.
  QString typeName = getTypeName(type);
  curve_Topology* T = getTopo(type);
  if (!T)
    return nullptr;
  curve_Curve* c = dynamic_cast<curve_Curve*>(T->child(typeName));
  // 2.
  if (!c)
    c = (curve_Curve*)TObject::new_object("curve_Curve", typeName, T);
  if (c)
    return c;
  return nullptr;
}

// 2.getCurve
QString InscribedCircleCalculator::getNurbsName(int type, int CurveID)
{
  QString NurbsName = "nurbs" + getTypeName(type) + QString::number(CurveID);
  return NurbsName;
}

curve_Nurbs* InscribedCircleCalculator::getNurbs(int type, int CurveID, bool createIfNotAvaiable)
{
  // 1.
  curve_Curve* Curve = getCurve(type);
  if (!Curve)
    return nullptr;
  QString NurbsName = getNurbsName(type, CurveID);
  curve_Curve* c = Curve->getCurveByName(NurbsName);

  // 2.
  if (!c && createIfNotAvaiable)
    c = Curve->addSegment(curve_Curve::Nurbs, NurbsName);
  if (c)
  {
    curve_Nurbs* s = dynamic_cast<curve_Nurbs*>(c);
    return s;
  }
  return nullptr;
}
```

InscribedCircleCalculator.cpp

```cpp
QString InscribedCircleCalculator::getCircleName(int type, int CurveID)
{
  QString CircleName = "Circle" + getTypeName(type = 0)
    + QString::number(CurveID);

  return CircleName;
}

curve_Circle* InscribedCircleCalculator::getCircle(int type, int CurveID, bool createIfNotAvaiable) { ... }

curve_Nurbs* InscribedCircleCalculator::getCenterCurve(int type, bool createIfNotAvaiable) { ... }

curve_Nurbs* InscribedCircleCalculator::getAreaCurve(int type, bool createIfNotAvaiable)
{
  // 1. T
  curve_Curve* Curve = getCurve(type);
  if (!Curve)
    return nullptr;
  QString AreaCurvename = "AreaCurve";
  curve_Curve* AreaCurve = Curve->getCurveByName(AreaCurvename);

  // 2.
  if (!AreaCurve && createIfNotAvaiable)
    AreaCurve = Curve->addSegment(curve_Curve::Nurbs, AreaCurvename);

  // 3.
  if (AreaCurve)
  {
    curve_Nurbs* ac = dynamic_cast<curve_Nurbs*>(AreaCurve);
    return ac;
  }
  else
    return nullptr;
}
```

InscribedCircleCalculator.cpp

```cpp
curve_Circle* InscribedCircleCalculator::getCircle(int type, int CurveID, bool createIfNotAvaiable)
{
  // 1.
  curve_Curve* Curve = getCurve(type);
  if (!Curve)
    return nullptr;
  QString CircleName = getCircleName(type, CurveID);
  curve_Curve* c = Curve->getCurveByName(CircleName);
  if (!c && createIfNotAvaiable)
    c = Curve->addSegment(curve_Curve::Circle, CircleName);
  // 2.
  if (c)
  {
    curve_Circle* cc = dynamic_cast<curve_Circle*>(c);
    return cc;
  }
  return nullptr;
}

curve_Nurbs* InscribedCircleCalculator::getCenterCurve(int type, bool createIfNotAvaiable)
{
  // 1. T
  curve_Curve* Curve = getCurve(type);
  if (!Curve)
    return nullptr;
  QString CenterCurvename = "CircleCenterCurve";
  curve_Curve* CenterCurve = Curve->getCurveByName(CenterCurvename);

  // 2.
  if (!CenterCurve && createIfNotAvaiable)
    CenterCurve = Curve->addSegment(curve_Curve::Nurbs, CenterCurvename);

  // 3.
  if (CenterCurve)
  {
    curve_Nurbs* ccc = dynamic_cast<curve_Nurbs*>(CenterCurve);
    return ccc;
  }
  else
    return nullptr;
}
```