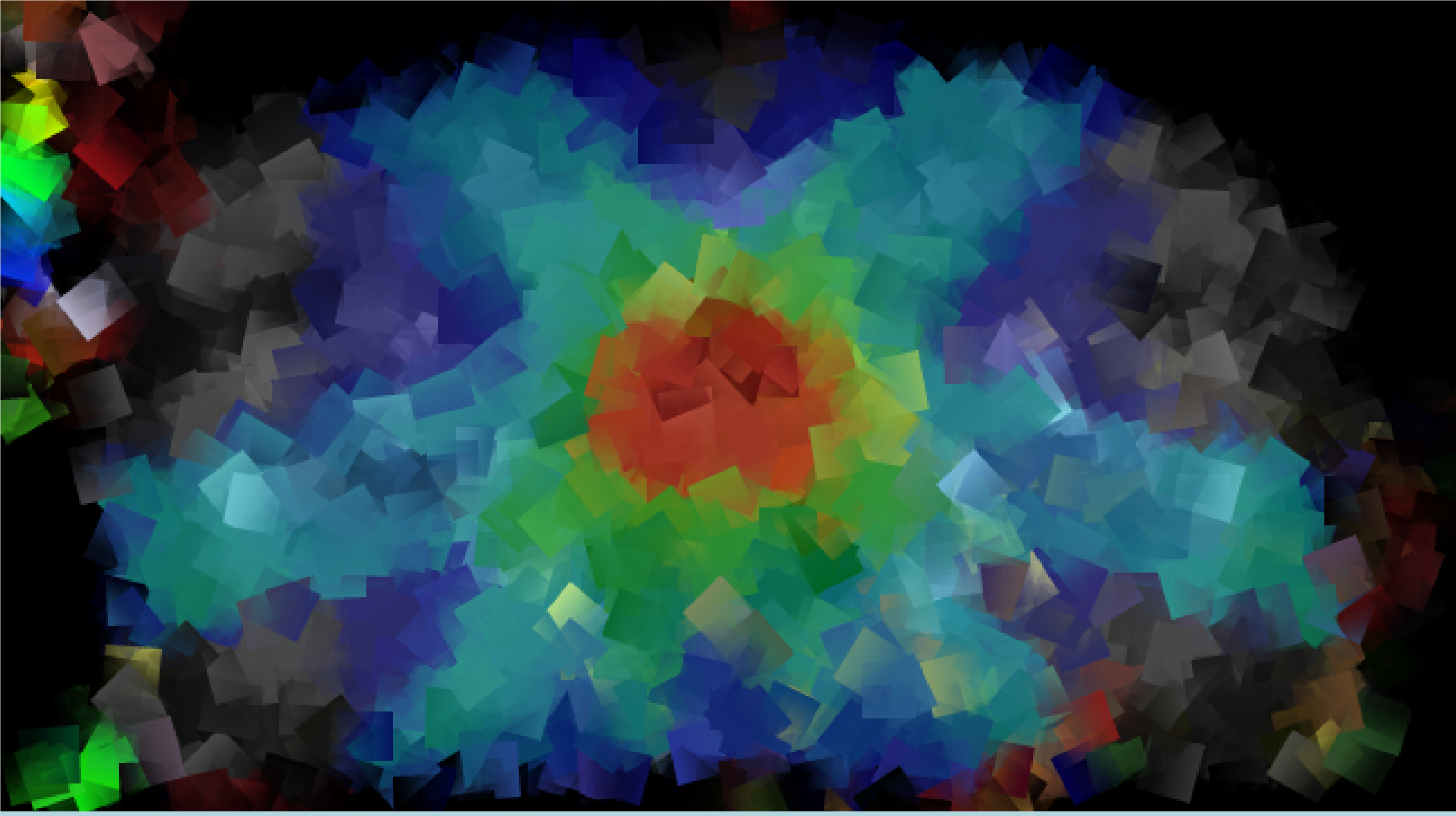
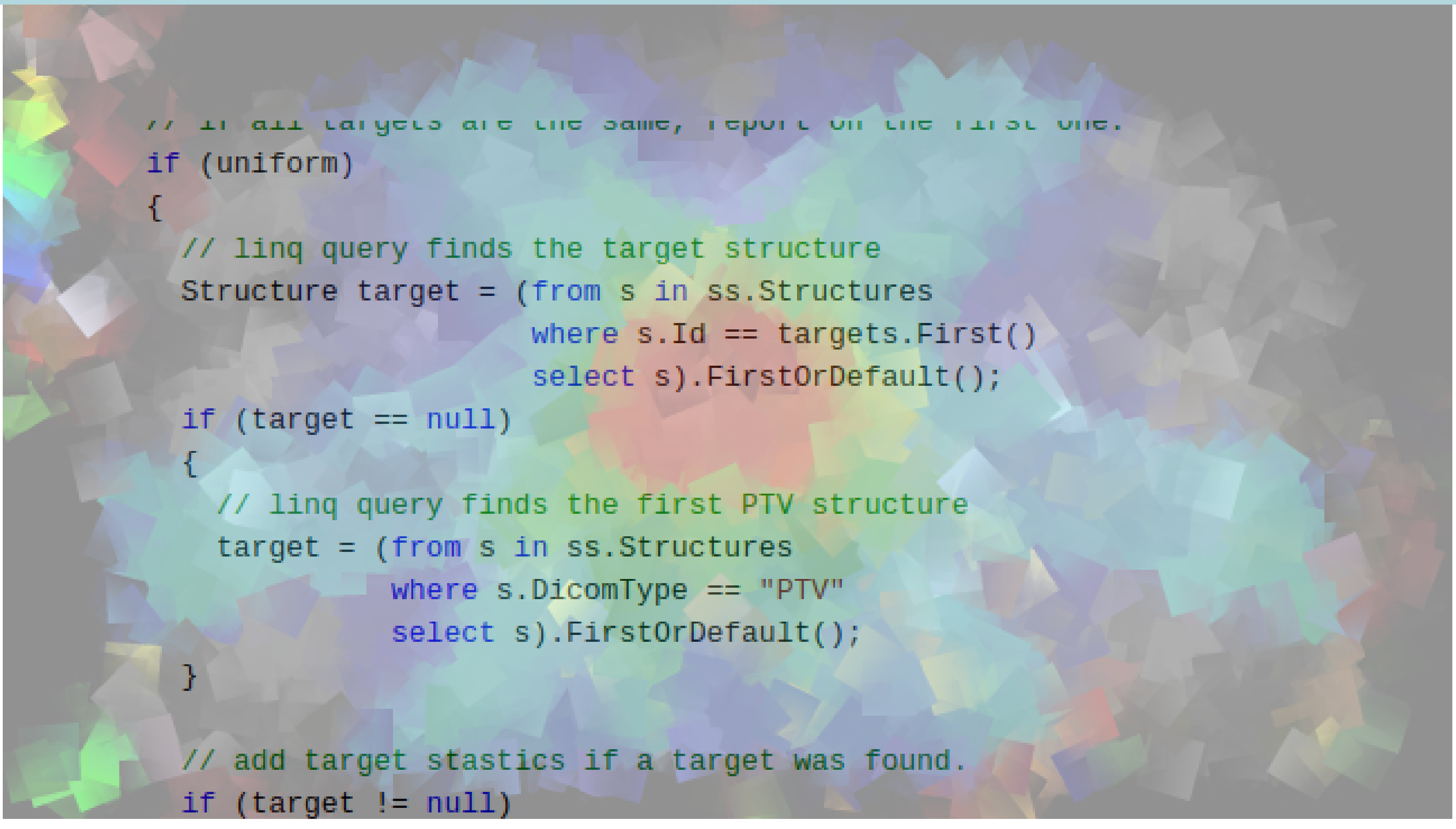


Varian APIs

A handbook for programming in the Varian oncology software ecosystem

Eds. Joakim Pyyry and Wayne Keranen



```
// if all targets are the same, report on the first one.
if (uniform)
{
  // linq query finds the target structure
  Structure target = (from s in ss.Structures
                      where s.Id == targets.First()
                      select s).FirstOrDefault();
  if (target == null)
  {
    // linq query finds the first PTV structure
    target = (from s in ss.Structures
              where s.DicomType == "PTV"
              select s).FirstOrDefault();
  }

  // add target stastics if a target was found.
  if (target != null)
```

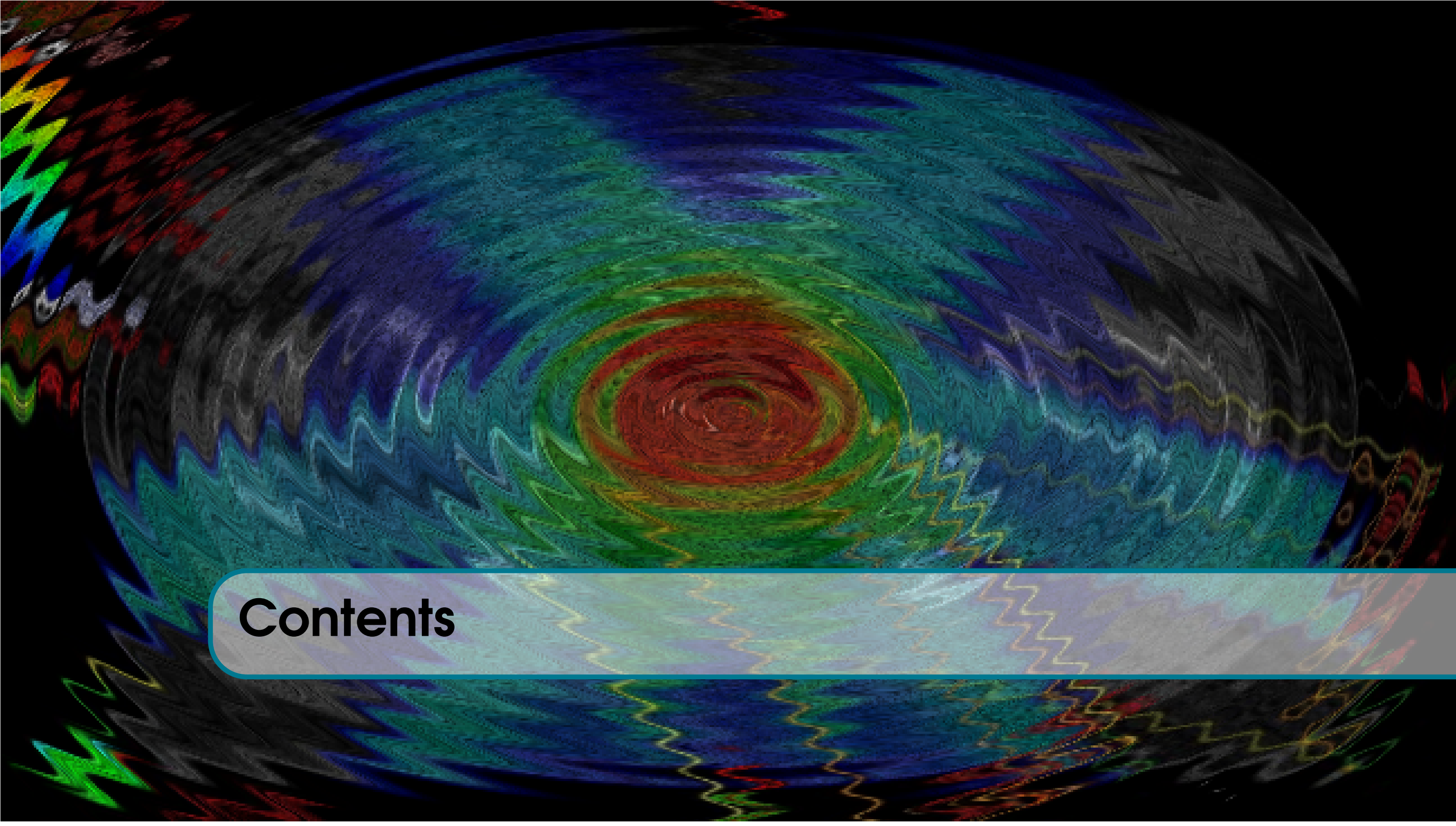
Copyright © 2018 Authors

The book is licensed under the Attribution-ShareAlike 4.0 International (CC BY-SA 4.0). You may obtain a copy of the license at `https://creativecommons.org/licenses/by-sa/4.0/`.

HTTP://WWW.VARIANDEVELOPER.COM/

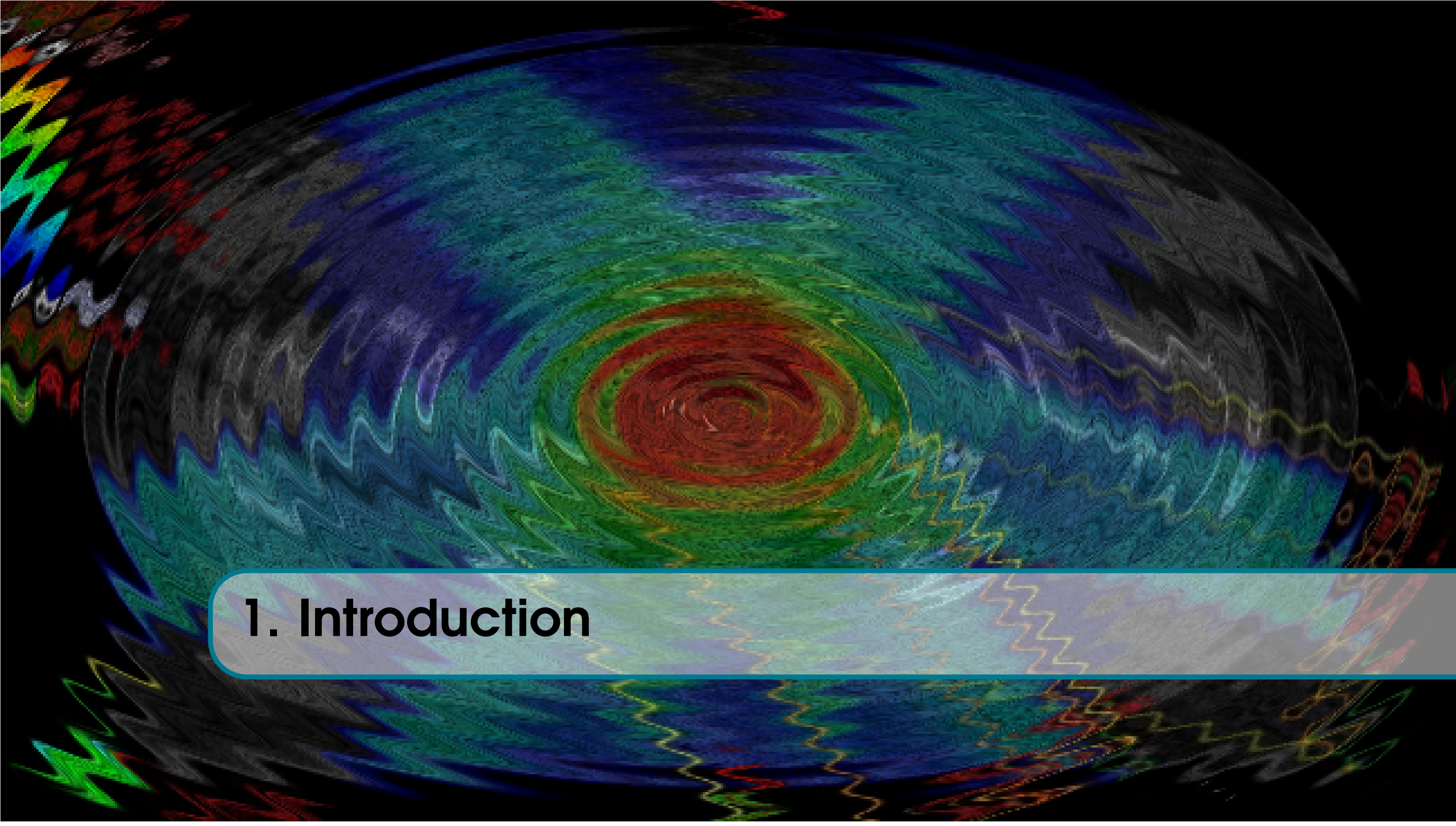
The book layout modified was modified form The Legrand Orange Book `https://www.overleaf.com/10515046znqvfjzghgyg#` licensed under the Creative Commons Attribution-NonCommercial 3.0 Unported License `http://creativecommons.org/licenses/by-nc/3.0`.

First edition v.0.9.0, July 2018



Contents

Part One



1. Introduction

Joakim Pyyry, D.Sc., Wayne Keranen

1.1 Background

Computing has been in the forefront of technology development of radiotherapy since the 1960s and continues to do so today. In the early days custom software for common purpose computers like the DEC PDP8 allowed research groups to develop treatment planning dose calculation programs and other tools for the clinic. Today we have commercial software systems, but the need to extend and develop custom software to augment these systems for research and other purposes still exists.

Physicist are used to model things mathematically, gather data and use computational tools to derive results based on models and data. Typical tools include spreadsheet software and mathematical software packages like MATLAB® or Mathematica®. In recent years using programming language Python with collection of packages known as SciPy has become popular for scientific computing. The Python ecosystem provides a broad range of open source software tools in many areas such as machine learning and interfacing to many software systems and packages.

In radiotherapy there are research software packages that are used broadly. One such package is a software environment called the computational environment for radiotherapy research (CERR, pronounced “sir”, `https://cerr.github.io/CERR/`) based on MATLAB® [7]. Other popular packages are Monte Carlo radiation simulation software suites such as EGSnrc, Geant and MCNPX as well as derivate works such as BEAMnrc [15] and Topas MC [13]. Many practitioners also utilize commercial treatment planning systems and their programming interfaces which are provided by most vendors.

The aim of this book is to provide practical guidance for software development for radiotherapy. The chapters will cover variety of topics with examples and code to get started with programming for your own needs. The content is geared towards the Varian software platform including the application programming interfaces, but provides also examples of using more common purpose open source tools useful for radiotherapy applications.

The high level overview of the Varian software platform is depicted in Figure 1.1. Varian applications store data in a central database which allows integration of applications and workflows.

The data is also aggregated into a data warehouse for reporting purposes. Data is exposed as a reporting data model from the data warehouse for customizable reporting purposes. Access to data is also provided via DICOM services as well as programming interfaces at the application level. Additional access is provided also via a few custom data services in ARIA Access.

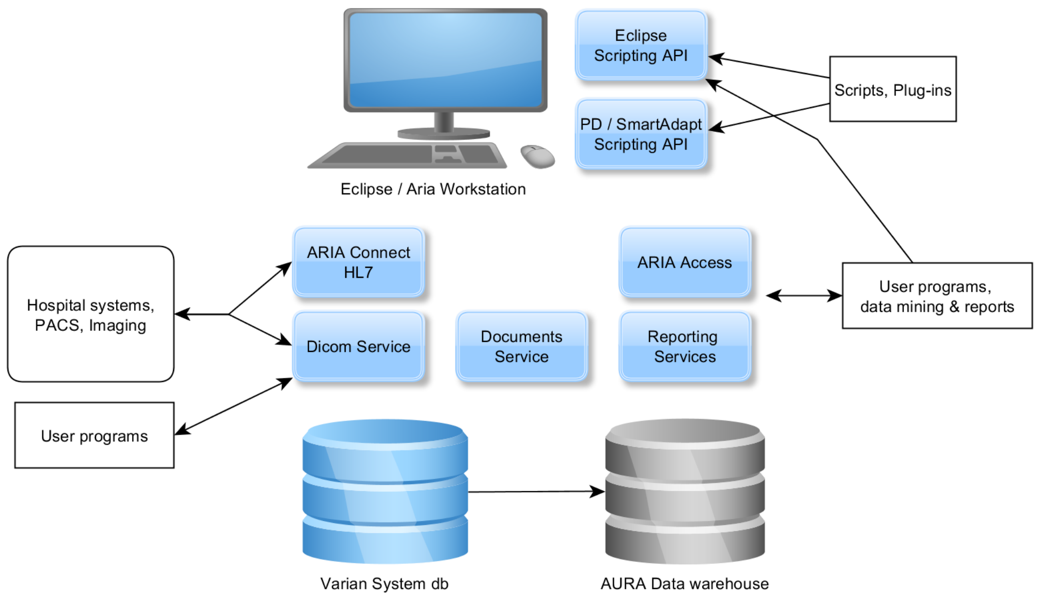


Figure 1.1: The Varian software platform contains multiple interface points on the data services side as well as on the application layer that allow access to data and extending the functionality of the system.

1.2 History of Varian's Developer Offering

Given the research-driven nature of the radiotherapy field and the close cooperation between Varian and its customer base there has always been a demand for Application Programming Interfaces (APIs) and other developer tools from the time software has been used in the field. Overview of various interfaces to access systems is shown in Table 1.1.

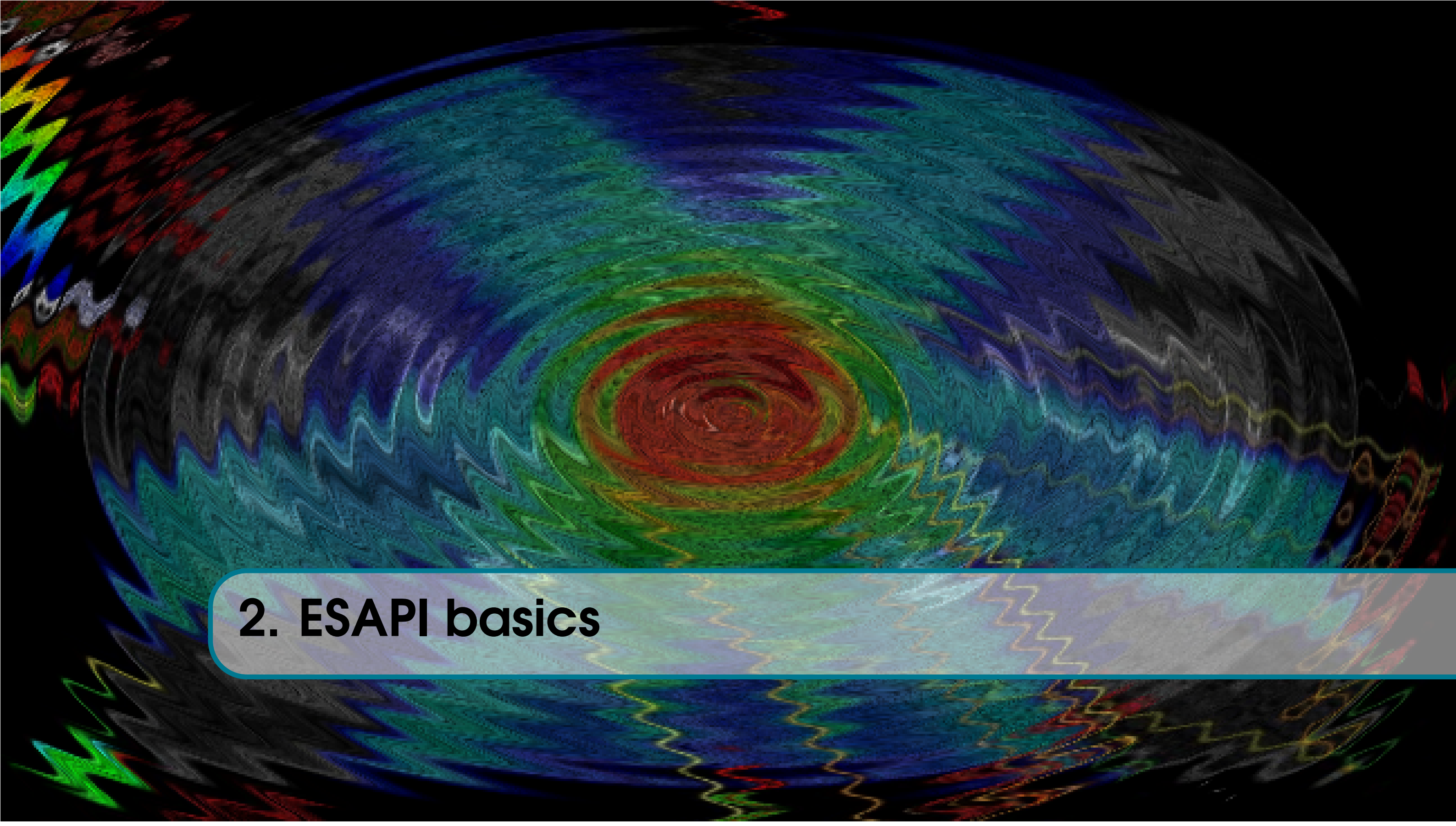
Before APIs data was often exchanged by third parties and Varian software using files. One popular file format was the MLC file format [4]. Later versions of this file format are still supported today for exchanging MLC leaf position data. The MLC file format is a text-based tag=value file format devised by Craig van Antwerp. A popular tool used to validate and manipulate the MLC file was MLC Shaper.

Image data exchange in DICOM [1] format has been available since the early 1990 between imaging systems and radiotherapy treatment planning systems. The specification was expanded with radiotherapy objects in 1997 and was thereafter supported gradually by treatment planning vendors. DICOM is a powerful data exchange format and there is a chapter in this book exploring more details on how to use and manipulate DICOM files.

ARIA LINK [2] was introduced in the mid-1990's as VARiS LINK and was the first officially supported API for Varian's software ecosystem. ARIA LINK was a stored procedure interface into the ARIA database. This interface allowed for read and write operations of Aria patient demographics, scheduling, RT Course, Prescription, and Plans. Support for ARIA LINK interface was deprecated in the v15.0 ARIA platform release. ARIA LINK functionality has not yet been completely replaced. It has been replaced partially by the ARIA Access Web Service and partially by DICOM Services.

Interfaces	**Usage**	**Year**
MLC file format	MLC control points	1995-
ARIA Link	Radiotherapy and treatment management data	1995-2017
DICOM	Medical image and radiotherapy data via file or network operations	1998-
ARIA Reports	Database access for reporting purposes	1998
ARIA IEM	HL 7 demographic and EMR data	2005
Eclipse Algorithm API	Customized dose calculation and optimization algorithms	2008
Eclipse Scripting API	Access to treatment planning and segmentation data	2012
SmartAdapt Scripting API	Access to segmentation and image registration data	2013
Portal Dosimetry Scripting API	Access to treatment records and images	2013
ARIA Documents	Web service access to documents	2014
ARIA Access	Web service access for EMR data	2016
AURA	Dataware house and custom reporting	2015

Table 1.1: Varian software interfaces



2. ESAPI basics

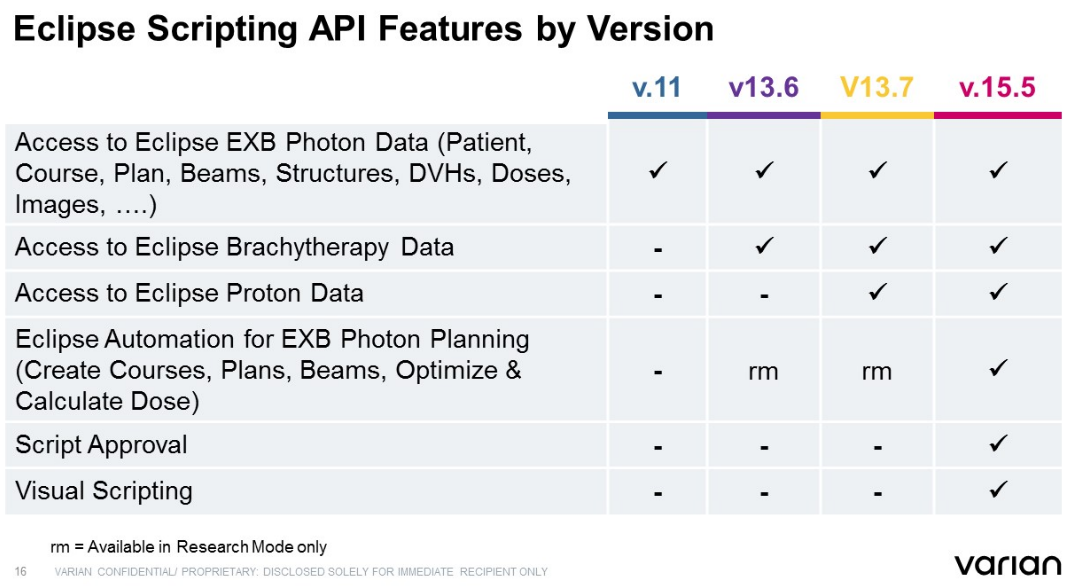
Wayne Keranen

2.1 What is ESAPI

2.1.1 Introduction

Eclipse Scripting API (ESAPI) is an Application Programming Interface (API) that is built into the Eclipse^{TM}treatment planning system. This API allows developers to create C#.NET scripts, DLLs, and programs that can read and operate on patient data loaded in Eclipse^{TM}, or on all patients in the Eclipse database. This API exposes data and algorithms for photon external beam, proton, and brachytherapy planning data, and allows modifications to photon external beam radiotherapy treatment plans.

Eclipse Scripting API was first released with Eclipse v11 as a read-only API that provided access to External Beam workspace data with an emphasis on allowing extraction of external beam photon treatment planning data, structure sets, 3D dose and image matrices, and DVH data. Major releases since then included v13.6, v13.7, and v15.5. In v13.6, RapidPlan, optimization support, and brachytherapy data model access was added. V13.7 added proton datamodel access. V15.5 added the Eclipse Automation feature set which includes clinically writable scripting and script approval. V15.5 also adds visual scripting, a new scripting mode that is a visual programming flow designer for non-programmers.



Eclipse Scripting API Features by Version

	v.11	v13.6	V13.7	v.15.5
Access to Eclipse EXB Photon Data (Patient, Course, Plan, Beams, Structures, DVHs, Doses, Images, ...)	✓	✓	✓	✓
Access to Eclipse Brachytherapy Data	-	✓	✓	✓
Access to Eclipse Proton Data	-	-	✓	✓
Eclipse Automation for EXB Photon Planning (Create Courses, Plans, Beams, Optimize & Calculate Dose)	-	rm	rm	✓
Script Approval	-	-	-	✓
Visual Scripting	-	-	-	✓

rm = Available in Research Mode only

16 VARIAN CONFIDENTIAL/ PROPRIETARY: DISCLOSED SOLELY FOR IMMEDIATE RECIPIENT ONLY

Eclipse Scripting API including automation features is a part of the Eclipse medical device and has been developed to meet the same global regulatory standards as the Eclipse treatment planning system.

Use of ESAPI is popular, especially in the United States where there are several hundred active scripters. The main uses of scripting to date have been in evaluating DVH metrics, reporting scripts, and plan checking assistant scripts [6].

Several small companies have created and are marketing Eclipse scripts for sale. Radformation (https://www.radformation.com/), for example, has received FDA 510K clearance for 2 different scripts; ClearCheck - a plan checking assistant script, and EasyFluence - an automated breast planning script. RedIon (https://www.redion.io/) is selling a DICOM Anonymizer script called DicomAnon. Varian has created an app store called Varian Marketplace (https://varian.force.com/vmarketlogin) where scripts and other commercial assets developed for the Varian ecosystem are distributed.

2.1.2 C#.NET

Eclipse Scripting API is implemented as a C#.NET class library. As such, ESAPI can be integrated into any Windows program which is .NET compatible, and ESAPI scripts can use .NET compatible libraries. The programming features available to ESAPI scripts are limited only by what the available .NET libraries provide, the rules Eclipse adds for scripting, and the security policies of the IT administrators at the site implementing scripts.

Eclipse adds a few restrictions for scripts. Eclipse internals are single threaded, so Eclipse adds a requirement that all ESAPI library calls from a script must be made from the same thread. With the script approval feature Eclipse adds additional restrictions pertaining to use of .NET reflection so that the script approval features cannot be circumvented.

Local security policies can add limitations to ESAPI capabilities. Local IT administrators may decide that the local disk cannot be written to, for example, and implement security restrictions accordingly so that any ESAPI script which writes to the local disk will fail.

2.1.3 ESAPI Runtime Modes

ESAPI can be used in 2 different modes of interaction - with plug-in scripts, and with standalone executable scripts. The API available in both modes is essentially the same with small differences in how patient context is established and accessed.

Plug-in Scripts

Plug-in scripts can be launched and run from the Eclipse user interface in the External Beam and Brachyvision workspaces. After launch, the plug-in is given access to the data of the currently open patient by Eclipse. Eclipse supports two types of plug-ins:

- **Single-file plug-in:** A source code file that Eclipse reads, compiles on the fly, and connects to the data model of the running Eclipse instance. Note that for v15.1.1 and later versions single-file plug-in scripts may not be usable on a clinical system; depending on the configuration. Write-enabled single file plug-in scripts can never be used on a clinical system, and read-only single-file plug-in scripts can only be used when Script Approval is not enforced for read-only scripts. The reason for this is that the Script Approval feature requires a version # to be compiled into the script and this is only possible by creating a DLL with the version # stored in the resource table of the compiled script DLL.
- **Binary plug-in:** A compiled .NET assembly that Eclipse loads and connects to the data model of the running Eclipse instance. Eclipse creates a Windows Presentation Foundation child window that the script code can then fill in with its own user interface components.

Plug-in scripts receive the current context of the running Eclipse instance as an input parameter. The context contains the patient, plan, and image that are active in Eclipse when the script is launched. Plug-in scripts can access data and operate on the active Eclipse patient only.

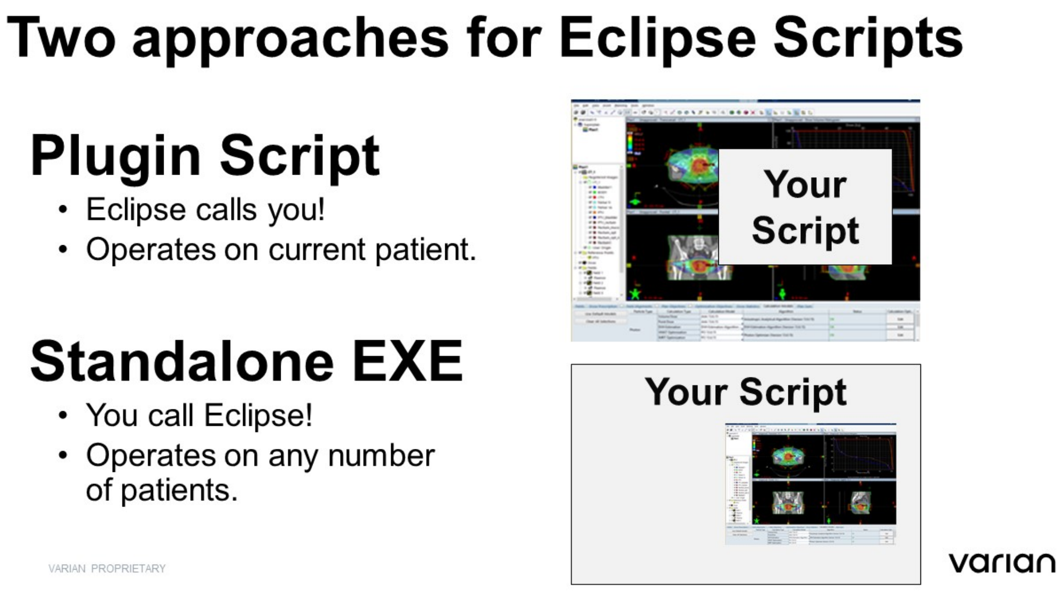
The following code sample is a plug-in script that extracts and displays the IDs for the active context items patient, course, plan, 3D image, and structure set.

Code 2.1
```
using System;
using System.Text;
using System.Windows;
using VMS.TPS.Common.Model.API;
using VMS.TPS.Common.Model.Types;

namespace VMS.TPS
{
  public class Script
  {
    public void Execute(ScriptContext context /*, System.Windows.Window
        window*/)
    {
        // Eclipse passes the application context to the script with
        // parameter 'ScriptContext context'
        // In this case the passed window is commented out since this
        // script does not use it.

        // Define variables to hold references to the active
        // patient, course, plan, 3D image, and structure set
        Patient patient = context.Patient;
        Course course = context.Course;
        PlanSetup plan = context.PlanSetup;
        Image image3D = context.Image;
        StructureSet structureSet = context.StructureSet;

        // format a string that shows the active context.
        string msg = string.Format(
            "Context:\n"+
            "\tPatient=\t\t{0}\n"+
            "\tCourse=\t\t{1}\n" +
            "\tPlan␣=\t\t{2}\n" +
            "\tImage=\t\t{3}\n" +
            "\tStructure␣Set␣=\t{4}\n",
            (patient != null) ? patient.Id : "not␣loaded",
                // null reference means the context object is not loaded
            (course != null) ? course.Id : "not␣loaded",
            (plan != null) ? plan.Id : "not␣loaded",
            (image3D != null) ? image3D.Id : "not␣loaded",
            (structureSet != null) ? structureSet.Id : "not␣loaded");
        MessageBox.Show(msg, "Varian␣Developer");
    }
  }
}
```

Plug-in scripts can be installed as a Favorite on the Eclipse Tools menu and assigned a shortcut key sequence to make it easy for users to run them.

Standalone Executable Scripts

A stand-alone executable script is a .NET application that uses the Eclipse Scripting API to gain access to Eclipse data and functions. It can be launched just like any Windows application. Stand-alone executables can be either command-line (console) applications, or they can use any .NET user interface technology available on the Windows platform. While plug-in scripts are restricted to work on the active loaded patient in Eclipse, stand-alone executables can scan the database and open any patient.



Two approaches for Eclipse Scripts

Plugin Script
- Eclipse calls you!
- Operates on current patient.

Standalone EXE
- You call Eclipse!
- Operates on any number of patients.

Your Script
Your Script
VARIAN PROPRIETARY
varian

Both plug-in and standalone executable scripts can have very sophisticated user interfaces since ESAPI uses C#.NET as its programming language. Frameworks like Microsoft Windows Presentation Foundation (WPF) help programmers build sophisticated UIs.

2.2 Getting started

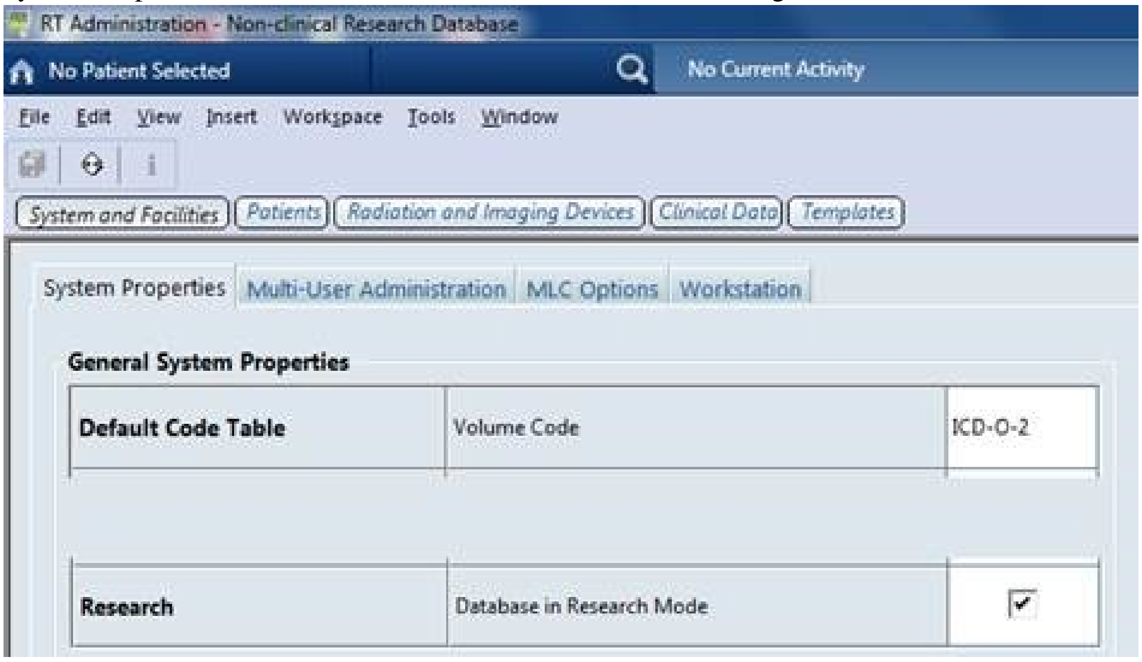
This section shows how to get started with the different Eclipse script types to gain access to Eclipse data and functions.

2.2.1 Developer System Setup and Configuration

Running Eclipse Scripts requires a computer workstation that has the Eclipse Treatment Planning System installed. The recommended development configuration is to have a non-clinical Eclipse system installed and configured for development purposes (ESAPI Dev System). Scripts are developed and tested on the ESAPI Dev System and then copied to the clinical environment when those scripts are ready for clinical evaluation or general clinical use.

Eclipse nonclinical development systems permit running unapproved write-enabled scripts. This insulates the clinical environment from development and removes the need to approve scripts during the program and test phase of development.

To configure an Eclipse system as an ESAPI Dev System (assuming you have the proper user rights and the Eclipse Scripting API for Research Users license): Navigate to the RT Administration Task using a system administrator account. Navigate to the System and Facilities workspace. Click the System Properties tab. Click the "Database in Research Mode" flag.



General System Properties

Default Code Table	Volume Code	ICD-O-2
Research	Database in Research Mode	✓

See the Eclipse Scripting API Reference Guide [3] for all the details. This guide can be accessed and downloaded by all Varian customers from MyVarian (https://myvarian.com/).

2.2.2 Developer Tools

The recommended configuration is to install your coding tools on the ESAPI Dev System. The most useful and best supported coding tool is Microsoft Visual Studio. Many of us use Visual Studio Professional, and there are other serious developers who use the Visual Studio Community Edition. Other tools that should be used for serious development are source code management tools and build / integration tools. Microsoft Team Foundation Server (TFS) and Git (https://git-scm.com/) are 2 popular options. Serious developers also use fully automated unit testing for their clinical software to keep future maintenance burdens low.

Debugging Plug-in Scripts

Debugging standalone executable scripts is easy - just compile a debug version of the script and run it within the Visual Studio debugger. Debugging plug-in scripts is less obvious.

There are two usual ways to debug plug-in scripts. The first way to debug a plug-in script is to develop the script within a project that uses a Plug-in Tester (https://github.com/VarianAPIs/samples/tree/master/Eclipse Scripting API/projects/PluginTester). The Plug-in Tester is a standalone executable script that provides a simple window that allows the user to establish the script context and pass it to the plug-in script. The plug-in script is modified slightly to fit into the Plug-in Tester framework. The Plug-in tester project is compiled in debug and the user runs the Visual Studio debugger on that project to debug the plug-in script.

The second way to debug a plug-in script is to run the script in Eclipse and attach the debugger to the Eclipse process.

Add Automated Unit Tests for Your Scripts

ESAPIX Facades (https://rexcardan.github.io/ESAPIX/articles/facades.html) allows for known data injection which is one way to automate your unit tests.

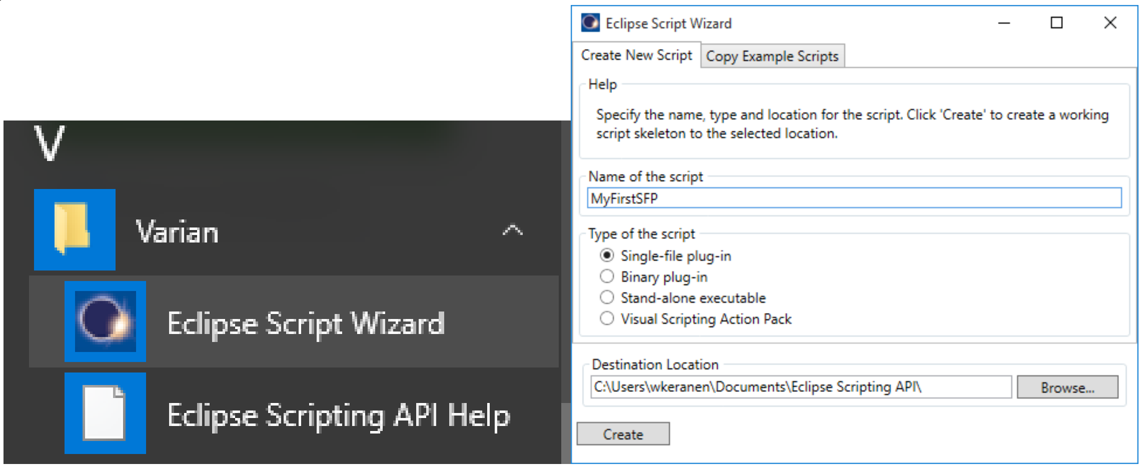
2.2.3 Your first script

The easiest way for the beginner to start scripting is to use the Eclipse Script Wizard to create a Visual Studio project that can be used for developing the script.

Single File Plugin

Note that this section works on an ESAPI Dev System and will not work for a 15.1 and higher Clinical system if script approvals are required for read-only scripts.

Select the Eclipse Script Wizard menu item from the Windows Start Menu to run the Script Wizard. When the Eclipse Script Wizard is running, type "MyFirstSFP" and choose "Single File Plug-in" as the script type. Choose an appropriate Destination Location for the Visual Studio project and click the Create button.



Eclipse Script Wizard

Create New Script | Copy Example Scripts

Help

Specify the name, type and location for the script. Click 'Create' to create a working script skeleton to the selected location.

Name of the script: MyFirstSFP

Type of the script
- ● Single-file plug-in
- ○ Binary plug-in
- ○ Stand-alone executable
- ○ Visual Scripting Action Pack

Destination Location: C:\Users\wkeranen\Documents\Eclipse Scripting API\ [Browse...]

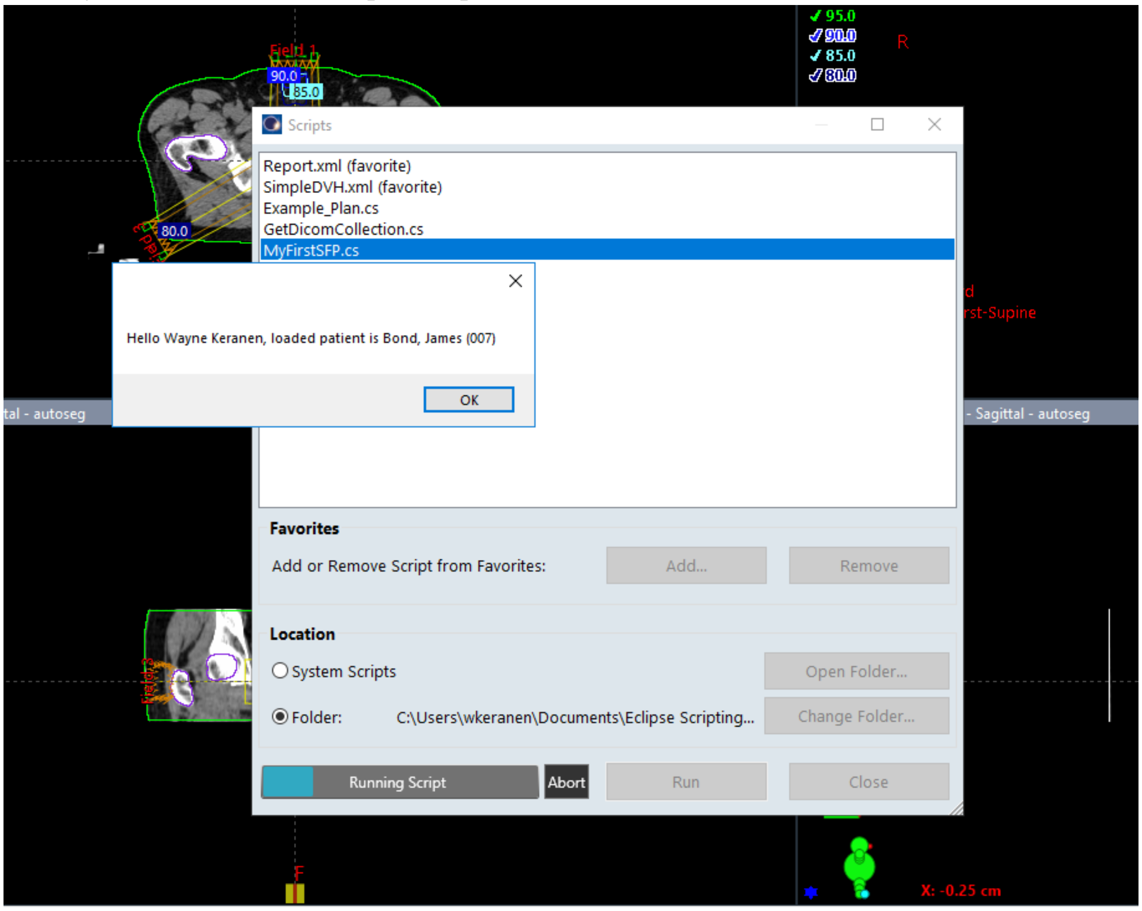
[Create]

Click "Yes" when prompted whether to launch Visual Studio after clicking Create. In the implementation of the Execute method, add a pleasant greeting message as shown in the following code listing.

Code 2.2
```
public void Execute(ScriptContext context /*, System.Windows.Window
    window, ScriptEnvironment environment*/)
    {
            // TODO : Add here the code that is called when the script
                is launched from Eclipse.
            MessageBox.Show("Hello␣" + context.CurrentUser.Name + ",␣
                loaded␣patient␣is␣" + context.Patient.Name);
    }
```

Save the code file and run Eclipse. Navigate to the External Beam workspace and load a patient.

Choose menu item Tools/Scripts, and navigate to and run the Single File Plugin script by choosing the file "MyFirstSFP.cs" in the Eclipse Script Wizard.



Report.xml (favorite)
SimpleDVH.xml (favorite)
Example_Plan.cs
GetDicomCollection.cs
MyFirstSFP.cs

Hello Wayne Keranen, loaded patient is Bond, James (007)

Folder: C:\Users\wkeranen\Documents\Eclipse Scripting...

2.3 ESAPI Features

2.3.1 Extract treatment planning data

ESAPI provides API methods to extract most treatment planning data that is available in Eclipse. To gain access to the needed data, the scripter starts with the ScriptContext for plug-in scripts and works down the object tree of the relevant object. For standalone executables, the scripter has to establish the context through other means, either by providing chooser dialogs so that the end user can identify and choose the relevant objects, or by implementing another scheme with a file that lists the context(s) to work on or similar.

2.3.2 Dose and Image Profiles

Introduction

Eclipse provides tools for visualizing arbitrary dose and image profiles on planes parallel with primary axis planes, that is either x,y, or z remain constant. Similar functionality is offered in ESAPI without the constraint on one of the dimensions. While profiles in Eclipse serve primarily as visualization tools they are much more potent in ESAPI. In ESAPI

1. Profiles are precise. Profile start and end points as well as resolution are all set as parameters.
2. Profiles are easily reproducible.
3. Profiles can be oriented to match any beam geometry.
4. Profiles are readily available for calculations.

API's

Dose profile

Code 2.3 — Dose Profile API.

```
public DoseProfile GetDoseProfile(VVector start,
    VVector stop,
    double[] preallocatedBuffer);
```

Arguments

start	3D-profile starting point in DICOM
stop	3D-profile ending point in DICOM
preAllocatedBuffer	Array of doubles

Return value

DoseProfile	uniform array of dose points on a line

Dose profile API samples the 3D dose uniformly along the line segment defined with `start` and `stop` such that first point is at `start` and last point is at `stop`. Dose values are tri-linearly interpolated at profile points.

Image profile

Code 2.4 — Image profile API.

```
public ImageProfile GetImageProfile(VVector start,
    VVector stop,
    double[] preallocatedBuffer);
```

Arguments

start	3D-profile starting point in DICOM
stop	3D-profile ending point in DICOM
preAllocatedBuffer	Array of doubles

Return value

ImageProfile	array of image points

Image profile API samples the 3D image uniformly along the line defined by `start` and `stop` such that first point is at `start` and last point is at `stop`.

Structure or segment profile

Code 2.5 — Image profile API.

```
public SegmentProfile GetSegmentProfile(
    VVector start,
    VVector stop,
    BitArray preallocatedBuffer);
```

Arguments

start	3D-profile starting point in DICOM
stop	3D-profile ending point in DICOM
preAllocatedBuffer	BitArray (boolean values)

Return value

Segment profile	vector of boolean values.

`GetSegmentProfile` returns an array of boolean values such that value is `true` for points inside the segment, `false` otherwise.

Examples

Image Profile

Following function returns a triple of image profiles through isocenter in primary axis directions.

Code 2.6

```
public static (ImageProfile, ImageProfile, ImageProfile)
    getImageProfilesThroughIsocenter(PlanSetup plan)
{
    var image = plan.StructureSet.Image;
    var dirVecs = new VVector[] {
        plan.StructureSet.Image.XDirection,
        plan.StructureSet.Image.YDirection,
        plan.StructureSet.Image.ZDirection
    };
    var steps = new double[] {
        plan.StructureSet.Image.XRes,
        plan.StructureSet.Image.YRes,
        plan.StructureSet.Image.ZRes
    };
    var planIso = plan.Beams.First().IsocenterPosition;
    var tmpRes = new ImageProfile[3];
    //
    // Throws if plan does not have 'BODY'
    //
    var body = plan.StructureSet.Structures.Single(st => st.Id=="BODY");
    for (int ind = 0; ind < 3; ind++)
    {
        (var startPoint, var endPoint) = Helpers.
            GetStructureEntryAndExit(
        body,
        dirVecs[ind],
        planIso, steps[ind]);

        var samples = (int) Math.Ceiling((endPoint - startPoint).Length
            / steps[ind]);
        tmpRes[ind] = image.GetImageProfile(startPoint, endPoint, new
            double[samples]);
    }
    return (tmpRes[0], tmpRes[1], tmpRes[2]);
}
```

Dose profile

Taking profile Image profiles may have little use outside reporting but dose profiles on the other hand can be leveraged for measurement comparison in beamline commissioning and quality assurance.

Dose profile API expects profile start and stop points in DICOM, which is not usually the case with measurements. The following example demonstrates how to sample a dose profile with start and stop points defined in gantry system. The returned profile points have their coordinates in DICOM but they can relatively easily be converted back. The example code only considers 4DOF couch and patient in HFS orientation.

Converting point from gantry to DICOM involves two rotations to account for gantry and patient support angles followed with a translation to account for isocenter.

Code 2.7 — Transform point in gantry to DICOM.

```
public static VVector GantryToDICOM(VVector point,
        double gantryInDegrees,
    double patientSupportInDegrees,
    VVector isoCenter)
{
        //
        // Account for gantry
        //
        var retval = RotateY(point, gantryInDegrees);
        //
        // Account for patient support
        //
        retval = RotateZ(retval, patientSupportInDegrees);
        //
        // Add beam isocenter and reassign axis (HFS patient)
        //
        return new VVector(retval.x, -retval.z, retval.y) + isoCenter;
}
```

where rotation about Y to account for gantry is

Code 2.8 — Rotation about Y .

```
public static VVector RotateY(VVector point,
        double angleInDeg,
    Direction dir = Direction.CW)
{
        var angleInRad = angleInDeg * 2 * Math.PI / 360;
        var c = Math.Cos(angleInRad);
        var s = ((int)dir) * Math.Sin(angleInRad);
        var x = point.x * c - point.z * s;
        var z = point.x * s + point.z * c;
        return new VVector(x, point.y, z);
}
```

and rotation about Z to account for patient support is

Code 2.9 — Rotation about Z.

```
public static VVector RotateZ(VVector point,
        double angleInDeg,
    Direction dir = Direction.CW)
{
        var angleInRad = angleInDeg * 2 * Math.PI / 360;
        var c = Math.Cos(angleInRad);
        var s = ((int)dir) * Math.Sin(angleInRad);
        var x = point.x * c - point.y * s;
        var y = point.x * s + point.y * c;
        return new VVector(x, y, point.z);
}
```

To get back into gantry coordinates just invert the operations

Code 2.10 — DICOM to gantry.

```
public static VVector DICOMToGantry(VVector point, double
    gantryInDegrees, double patientSupportInDegrees, VVector isoCenter)
{
        var retval = point - isoCenter;
        retval = new VVector(retval.x, retval.z, -retval.y);
        retval = RotateZ(retval, -patientSupportInDegrees);
        return RotateY(retval, -gantryInDegrees);
}
```

With tools to convert point in gantry system to DICOM, arbitrary dose profile with limits in gantry can be extracted with

Code 2.11 — Dose profile.

```
public static DoseProfile getBeamDoseProfile(
   VoiBox calculationVolume,
   Beam beam,
   VVector startInGantry,
   VVector stopInGantry,
   double stepSizeInmm = 2.5)
{
        //
        //Convert limits from gantry to DICOM
        //
        var start = Helpers.GantryToDICOM(startInGantry, beam.
            ControlPoints.First().GantryAngle, beam.ControlPoints.First()
            .PatientSupportAngle, beam.IsocenterPosition);
        var stop = Helpers.GantryToDICOM(stopInGantry, beam.
            ControlPoints.First().GantryAngle, beam.ControlPoints.First()
            .PatientSupportAngle, beam.IsocenterPosition);

        var dose = beam.Dose;

        //
        // Limit the requested profile with calculation volume
        // This step can be skipped, points falling outside dose will
        // be set as NaN
        //
        Helpers.cutWithCalculationVolume(calculationVolume, ref start,
            stop - start);
        Helpers.cutWithCalculationVolume(calculationVolume, ref stop,
            stop -start);
        //
        // Get the profile
        //
        return dose.GetDoseProfile(start, stop, new double[(int)Math.
            Ceiling((stop - start).Length / stepSizeInmm)]);
}
```

If exact spacing is required, `start` and `stop` must be tweaked for profile length which is multiple of the specified step size. This can for example be done as follows

Code 2.12 — Limit adjustment for exact spacing.

```
var profileSpan = (stop - start);
var spanLength = profileSpan.Length;
var fullStepLength = Math.Floor(spanLength/stepSizeInmm)*stepSizeInmm;
var adjust = stepSizeInmm - (spanLength - fullStepLength);
profileSpan.ScaleToUnitLength();
start = start - profileSpan * adjust * 0.5;
stop = stop + profileSpan * adjust * 0.5;
```

, which moves both limits by the same amount. However owing to nature of floating point numbers and calculations even adjusted limits may result in slightly off result.

Profiles may also be parameterized in terms other than start and stop positions in specified coordinate system. The following function for example defines the profile in terms of orientation, length, and offset from isocenter along beam axis.

Code 2.13 — Profile about beam axis.

```
public static DoseProfile getBeamDoseProfile(
        VoiBox calculationVolume,
        Beam beam,
        VVector OrientationInGantry,
        double profileMaxLength = 600,
        double distanceFromIsocenterTowardSourceInmm = 0,
        double stepSizeInmm = 2.5)
{

        if (OrientationInGantry.Length < 1e-8 || OrientationInGantry.
            Length > 1e8)
        {
                throw new ArgumentException("Invalid␣direction␣vector␣
                    for␣profile");
        }
        OrientationInGantry.ScaleToUnitLength();

        var start = -(profileMaxLength * .5) * OrientationInGantry;
        start.z += distanceFromIsocenterTowardSourceInmm;

        var stop =  (profileMaxLength *.5) * OrientationInGantry;
        stop.z += distanceFromIsocenterTowardSourceInmm;

        return getBeamDoseProfile(calculationVolume, beam, start, stop,
            stepSizeInmm);
}
```

Comparing data Reading in measurement input data is dependent on data format. For example a simple text file of comma separated values can be read in with the following two liner

Code 2.14 — Read in comma separated values.

```
var contents = File.ReadAllText("prof.csv").Trim().Split('\n');
prof3d = (from line in contents select line.Trim('\r').Split(',').Select
    (v => Double.Parse(v)).ToArray()).ToArray();
```

The code segment relies on c# feature linq to process input. The first line reads the entire content of a text file into a single string and splits it into an array of strings at line breaks. At this point `contents` is an array of strings of comma separated values for location and measured value. Second line further splits each line at commas and converts each value into double precision floating point number, ultimately returning an array of arrays `double[][]`

Since calculated points are taken along a line, profile can be flattened into a function of one argument.

Code 2.15 — Flatten profile.
```
public static double[][] flattenProfile(DoseProfile profile)
{
        var start = profile.ElementAt(0).Position;
        return profile.Select(pp => new double[] {(pp.Position - start).
            Length, pp.Value}).ToArray();
}
```

The function picks each profile point, calculates its distance from the start, and pairs it with corresponding dose value for an entry in the flattened 1D profile.

Original 3D profile points can be recovered with two points on the original profile.

Code 2.16 — Unflatten profile.
```
public static (VVector position, double value)[] unflattenProfile(double
    [][] flatProfile, VVector start, VVector stop)
{
        var dirVec = (stop - start);
        dirVec.ScaleToUnitLength();
        return flatProfile.Select(val => (start + val[0] * dirVec, val
            [1])).ToArray();
}
```

Measurement data, if taken along a line, can be similarly flattened. If the start and stop points were set with first and last measured dose values all that is required

Code 2.17 — Flatten measurement data.
```
var flatProfile = measurement.Select(val => new double[] { (new VVector(
    val[0], val[1], val[2]) - start).Length, val[3] }).ToArray();
```

where measurement is an array of `double[4]`. First three indices, 0 through 2, are for coordinates and the last is for measured value. If the start position was adjusted, the adjustment has to be accounted for as on offset from zero.

With calculated and measured data in the same space comparison can be done point by point with.

Code 2.18 — Calculate point wise difference between measured and calculated data.
```
private class MyCMP:IComparer
{
        public int Compare(object _a, object _b)
        {
                var a = _a as double[];
                var b = _b as double[];
                return a[0].CompareTo(b[0]);
        }
}

public static double[] distanceVec(DoseProfile doseProfile, double[][]
    measurements, IDifferenceCalculator diffCalc)
{
        //
        // If measurements are in 3D flatten into 1D.
        //
        var tmpMeas = measurements;
        if (measurements[0].Length == 4)
        {
                tmpMeas = measurements.Select(val => new double[] {(new
                    VVector(val[0], val[1], val[2]) - doseProfile[0].
                    Position).Length, val[3] }).ToArray();
        }
        //
        // Put measurements in increasing order on position
        // along profile, permutation is returned in
        // indices (this is usually redundant)
        //
        var indices = Enumerable.Range(0, tmpMeas.Length).ToArray();
        Array.Sort(tmpMeas, indices, new MyCMP());
        //
        // Flatten dose profile into 1D
        //
        var flatProfile = flattenProfile(doseProfile);
        //
        // Since measurement points are in increasing order,
        // we only need to march through profile once
        //
        var distVec = new double[tmpMeas.Length].Select(v => v = Double.
            NaN).ToArray();
        var profileIndex = 0;
        var measIndex = 0;
        foreach (var meas in tmpMeas)
        {
                //
                // step along profile until we have passed location of
        // the current measurement (meas)
                //
                for (; (profileIndex < flatProfile.Length) &&
        (flatProfile[profileIndex][0] < meas[0]); profileIndex++) ;
                //
                // Have we moved beyond calculated profile, if so bail
                    out
                //
                if (profileIndex == flatProfile.Length)
                        break;
                //
                // Calculate distance values with provided calculator
                //
                distVec[indices[measIndex++]] = diffCalc.Calculate(meas,
                     flatProfile, profileIndex);
        }
        return distVec;
}
```

Where `diffCalc` calculates the difference for the measured point in `meas`.

For example

Code 2.19 — Calculate difference between measured and calculated data.

```
class PointDifference:IDifferenceCalculator
{
    private double linInterp(double v1, double v2, double x)
    {
        return (1 - x) * v1 + x * v2;
    }

    private double DoseDiff(double[] profilePoint1, double[]
        profilePoint2, double[] measurement)
    {
        var len = (profilePoint2[0] - profilePoint1[0]);
        //
        // degenerate case, segment has zero length,
        //
        if (Math.Abs(len) < Double.Epsilon)
        {
            if (Math.Abs(profilePoint2[1]-profilePoint1[1]) < Double.
                Epsilon &&
            Math.Abs(measurement[0]-profilePoint1[0]) < Double.Epsilon)
            {
                return profilePoint1[1];
            }
            //
            // Either profile is bad, it has two values for
            // the same argument or measurement was not taken
            // at the profile point.
            //
            return Double.NaN;
        }
        var x = (measurement[0] - profilePoint1[0]) / len;
        //
        // does measurement fall in segment, if not return NaN
        //
        if (x < 0 || x > 1)
        {
            return Double.NaN;
        }
        var interpolatedDose = linInterp(profilePoint1[1], profilePoint2
            [1], x);
        return measurement[1] - interpolatedDose;
    }
    public double Calculate(double[] measurement,
                            double[][] profile, int profileIndex)
    {
        return DoseDiff(profile.ElementAt(profileIndex - 1), profile.
            ElementAt(profileIndex), measurement);
    }
}
```

compares measured dose with linearly interpolated calculated value at same point. Alternatively

Code 2.20 — Unscaled 1D gamma calculator.

```
class Unscaled1DGamma : IDifferenceCalculator
{
        public int Neighborhood { get; private set; } = 5;
        private double[] nhoodValues;

        public Unscaled1DGamma(int nhood = 5)
        {
                nhoodValues = new double[(Neighborhood = nhood)*2+1];
        }

        private double DotProd(double[] v1, double[] v2)
        {
                return v1.Zip(v2, (a, b) => (a * b)).Aggregate((a, b) =>
                     (a + b));
        }

    //
    // Shortest distance from measurement to line segment
    //
        private double MinDistance(double[] profilePoint1,
                                   double[] profilePoint2,
                                   double[] measurement)
        {
                //
                // a = profilePoint2 - profilePoint1
                //
                var a = profilePoint1.Zip(profilePoint2, (av, bv) => (bv
                     - av)).ToArray();
                //
                // b = measurement - profilePoint1
                //
                var b = profilePoint1.Zip(measurement, (av, bv) => (bv -
                     av)).ToArray();
                //
                // |a|
                //
                var aNorm = Math.Sqrt(DotProd(a, a));
                //
                // |b|
                //
                var bNorm = Math.Sqrt(DotProd(b, b));
                //
                // a * b
                //
                var aDotb = DotProd(a, b);
                //
                // scalarProjection = cos(alpha) * |b| = ((a*b)/(|a||b|)
                    )*|b| = (a*b)/|a|
                //
                var proj = aDotb / aNorm;
                //
                // measurement closest to segment start (profilePoint 1)
                //
                if (proj < 0)
                {
                        return bNorm;
                }
                //
                // projection > |a| => closest point is segment end
                    point (profilePoint2)
                //
                if (proj > aNorm)
                {
                        var c = profilePoint2.Zip(measurement, (av, bv)
                            => (bv - av)).ToArray();
                        return Math.Sqrt(DotProd(c, c));
                }
                //
                // Projection falls on line segment between
                    profilePoint1 and profilePoint 2;
                //
                return Math.Sqrt(bNorm * bNorm - proj * proj);
        }
    //
    // Consider segments in the neighborhood of the segment including
    // measurement argument.
    //
        public double Calculate(double[] measuredPoint, double[][]
            profile, int profileIndex)
        {
                var lInd = 0;
                var startInd = (int)Math.Max(profileIndex - Neighborhood
                    , 1);
                var endInd = (int)Math.Min(profileIndex + Neighborhood,
                    profile.Length);
                endInd = Math.Max(endInd, startInd+1);

                for (var nInd = startInd; nInd < endInd; nInd++, lInd++)
                {
                        nhoodValues[lInd] = MinDistance(profile[nInd -
                            1], profile[nInd], measuredPoint);
                }
                var minVal = nhoodValues.Take(lInd).Min((v) => { return
                    Double.IsNaN(v) ? Double.PositiveInfinity : Math.Abs(
                    v);});
                return double.IsPositiveInfinity(minVal) ? double.NaN :
                    minVal;
        }
}
```
■

calculates the shortest distance from a measurement point to a piecewise linear calculated profile.

Putting it all together With the methods in the previous paragraph extracting dose profile and comparing it with the corresponding measurement becomes

Code 2.21
```
//
// 'Measured data' in TData.doseProfile
//
var first = TData.doseProfile.First();
var last = TData.doseProfile.Last();
start = new VVector(first[0], first[1], first[2]);
stop = new VVector(last[0], last[1], last[2]);
//
// Pick beam
//
var beam = planSetup.Beams.ElementAt(4);
var cProf = Profile.getBeamDoseProfile(
        planSetup.GetCalculationVolume(),
    beam,
    start,
    stop, 1.0);

//
// flatten 'measurement'
// note, 'start' is the same point albeit in different coordinate
// system as 'start' in 'cProf'
//
var dp = TData.doseProfile;
var flatProfile = dp.Select(val => new double[] {
        (new VVector(val[0], val[1], val[2]) - start).Length, val[3] }).
            ToArray();
//
// get vector of differences (distance is bit of misnomer, as
// the value my be signed.)
//
var dv = Helpers.distanceVec(cProf, flatProfile, new Unscaled1DGamma());
```

Considerations The example presented above compares measured data only to calculated data along the measurement path. It makes no provisions for uncertainties and noise in the measurement process which especially in high gradient areas may be enough to throw the result off. A relatively simple way to get a handle on the uncertainties is to consider nearby dose profiles taken for example in 3-neighborhood of the measured profile. A more robust way is compare doses with 3D gamma index, calculation of which is beyond scope of this section.

Dose plane

Dose profiles can also be leveraged to construct rectangular dose planes by scanning in direction of an edge of the rectangle and taking dose profiles in direction of the other. Following method extracts an orthogonal to beam dose plane at the set offset from isocenter.

Code 2.22 — Extract orthogonal dose plane.

```
public static double[,] OrthogonalDoseCrossSection(VoiBox calcVol,
      Beam beam,
      double isocenterOffset= 0,
      double xSizeInmm = 200,
      double zSizeInmm = 200,
      double pixelSizeInmm = 2.5)
{
    //
    // Plane X axis orientation
    //
    var xDir = Helpers.GantryToDICOM(new VVector(1, 0, 0), beam.
        ControlPoints.ElementAt(0).GantryAngle, beam.ControlPoints.
        ElementAt(0).PatientSupportAngle, new VVector(0,0,0));
    //
    // Beam axis direction
    //
    var normVec = Helpers.directionTowardSource(beam);
    //
    // Individual dose profile X start and end positions on plane
    //
    var start = beam.IsocenterPosition - xSize * xDir * 0.5;
```

```
        var end = beam.IsocenterPosition + xDir * 0.5;
        start = start + normVec * isocenterOffset;
        end = end + normVec * isocenterOffset;
        Helpers.cutWithCalculationVolume(calcVol, ref start, xDir);
        Helpers.cutWithCalculationVolume(calcVol, ref end, xDir);
        //
        // Plane Z axis orientation
        //
        var zDir = Helpers.CrossProduct(xDir, normVec);
        //
        // These must be zero, otherwise
        // something has gone pot.
        //
        var a = zDir.ScalarProduct(xDir);
        var b = xDir.ScalarProduct(normVec);
        //
        // Z limits
        //
        var zStart = beam.IsocenterPosition + normVec * isocenterOffset  -
            zDir * zSize * 0.5;
        var zEnd = zStart + zDir * zSize;
        Helpers.cutWithCalculationVolume(calcVol, ref zStart, zDir);
        Helpers.cutWithCalculationVolume(calcVol, ref zEnd, zDir);

        var dose = beam.Dose;

        var xSamples = (int) Math.Ceiling((end-start).Length/ pixelSize);
        var zSamples = (int) Math.Ceiling((zEnd - zStart).Length/pixelSize);
        var tmpResult = new double[zSamples * xSamples];

        var rowIndex = 0;

        start -= zSamples / 2 * zDir * pixelSize;
        end -= zSamples / 2 * zDir * pixelSize;

        for (double zPos = 0;  zPos < zSamples; zPos++)
        {
            start += zDir * pixelSize;
            end += zDir * pixelSize;
            var prof = dose.GetDoseProfile(start, end, new double[xSamples])
                ;
            //
            // Replacing NaN's with 0's may not be smartest thing, NaN's are
                for unknown not 0 dose
            //
            var pvals = prof.Select(point => Double.IsNaN(point.Value) ? 0 :
                point.Value).ToArray();
            Array.ConstrainedCopy(pvals, 0, tmpResult, rowIndex * xSamples,
                xSamples);
            rowIndex++;
        }
        var result = new double[zSamples, xSamples];
        Buffer.BlockCopy(tmpResult, 0, result, 0, zSamples * xSamples *
            sizeof(double));
        return result;
}
```

If the dose plane is compared against measured data, methods described for 1D case, profile, can be extended for 2D case.

DVH calculation

Eclipse does not have histogram calculation for BED, EQD2 or similar derived values. For a single plan DVH equivalent histograms can be calculated efficiently by recalculating bin limits and re-sampling the normal DVH. However, as DVH preserves no spatial information, calculating histogram over sum of dose values, each of which is to be independently converted, cannot be done in this manner. Instead contributing dose matrices must be scanned separately, dose values converted to desired representation and accumulated with other values.

The following function creates a histogram over sum of converted dose values. Dose value conversion is passed into the function as an object argument `IDoseValueConverter converter`.

Code 2.23 — Calculate plan sum DVH for a structure.

```
public static (DVHBin[], DVHBin[], double) StructureDVH(
    Dose[] doses,
    Structure structure,
    IDoseValueConverter converter, int bins = 1024)
{
    var ddvh = new int[bins];
    //
    // Add epsilon to make sure every dose value is below upper limit of
         last bin.
    //
    var max = converter.Convert(doses.Sum(ds=>ds.DoseMax3D.Dose)) +
        Double.Epsilon;
    var step = max / bins;
    //
    // Contains structure, used to minimize scanned volume.
    //
    var boundingBox = structure.MeshGeometry.Bounds;
    //
    // Length of individual dose profile
    //
    var profileSamples = (int) Math.Ceiling(boundingBox.SizeX / doses
        [0].XRes);
    //
    // number of voxels in strucutre
    //
    var counter = 0;
    //
    // Scan in Z
    //
    for (var z = boundingBox.Z; z < boundingBox.Z + boundingBox.SizeZ; z
         += doses[0].ZRes)
    {
        //
        // Scan in Y
        //
        for (var y = boundingBox.Y; y < boundingBox.Y + boundingBox.
            SizeY; y += doses[0].YRes)
        {
            var start = new VVector(boundingBox.X, y, z);
            var stop = new VVector(boundingBox.X + boundingBox.SizeX, y,
                z);
            var sumOfConvertedDoseValues = new double[profileSamples];
            //
            // Loop over 'doses' and accumulate converted dose values
            //
            foreach (var dose in doses)
            {
                //
                // get vector of converted dose values along profile
                //
                var convertedDoseValues = dose.GetDoseProfile(start,
                    stop, new double[profileSamples]).Select(dv =>
                    converter.Convert(dv.Value)).ToArray();
                for (var index= 0; index < sumOfConvertedDoseValues.
                    Length; index++)
                {
                    sumOfConvertedDoseValues[index] +=
                        convertedDoseValues[index];
                }
            }
            //
            // Get structure profile to determine whether a dose point
            // is inside or outside of 'structure'
            //
            var structureProfile = structure.GetSegmentProfile(start,
                stop, new BitArray(profileSamples)).Select(profilePoint
                => profilePoint.Value).ToArray();
            for (var ind = 0; ind < structureProfile.Length; ind++)
            {
                if (true == structureProfile[ind])
                {
                    counter++;
                    var bin = (int)Math.Floor(sumOfConvertedDoseValues[
                        ind] / step);
                    ddvh[bin]++;
                }
            }
        }
    }
    //
    // Generate output data, for differential 'DVH' value is divided
    // by bin width (step) for result that is comparable with Eclipse.
    //
    var voxelVol = doses[0].XRes * doses[0].YRes * doses[0].ZRes * 1e-3;
    var voxelVolOverStep = voxelVol / step;
    var sampleCoverage = (counter * voxelVol/structure.Volume);
    var diffDVH = new DVHBin[bins];
    var cumDVH = new DVHBin[bins];
    var binCenter = step / 2;
    var val = counter;
    for (var ind = 0; ind < bins; ind++, binCenter += step)
    {
        diffDVH[ind] = new DVHBin() { doseValue = binCenter, Volume =
            ddvh[ind] * voxelVolOverStep};
        cumDVH[ind] = new DVHBin() { doseValue = binCenter, Volume = (
            val -= ddvh[ind]) * voxelVol};
    }
    return (cumDVH, diffDVH, sampleCoverage);
}
```

■

where dose converter can for example be defined as

Code 2.24 — EQD2 dose value converter.

```
public class EQD2:IDoseValueConverter
```

```
{
    public double alpha { get; set; }
    public double beta { get; set; }
    public int fractionNumber { get; set; }
    public double Convert(double val)
    {
        return val * ((val / fractionNumber) * (alpha / beta) / (2 + (
            alpha / beta)));
    }
}
```

where α and β are structure or organ dependent model parameters. The method scans segment and dose volumes in two dimensions, takes profiles in third and accounts for dose values which are inside the segment.

The method works reasonably well for large structures with relatively small number of voxels on the structure boundary but is increasingly inaccurate with small structures. This is because the method treats each voxel as entirely inside or outside. Consequently accuracy of the method can be improved at the expense of computational cost by sampling the dose and structure volume at finer resolution.

2.3.3 Automation

Starting in version 15, you can create scripts that modify RT data. With this feature, it is easy to automate some repetitive tasks, starting from structure creation and plan generation, optimization and dose calculation all the way to plan QA.

Writable scripts need to be approved following the institution's guidelines. Eclipse provides a tool to approve scripts for evaluation (where for a period of time, only certain people can run the script). After final approval, the script is in general use. The read-only scripts can be used without approval, or if so decided, must go through the same approval process. See Chapter 2.3.6 for details on script approval.

A couple of things need to be present for a writable script:

The Eclipse Script Wizard in version 15.5 will create the following line for you in the main script file, outside any namespace or class:`[assembly: ESAPIScript(IsWriteable = true)]`

Before the patient data is modified, this method needs to be called:

Code 2.25
```
Patient patient = context.Patient;
patient.BeginModifications();
```

After modification in a standalone script, this method needs to be called to save the changes:

Code 2.26
```
Application app = ...
app.SaveModifications();
```

For a plugin script, user saves the modifications in the Eclipse UI after the script has finished.

Optimization Structure Creation

Use margin function to create an optimization structure:

Code 2.27
```
StructureSet ss = context.StructureSet;
if (ss.CanAddStructure("CONTROL", "PTV+5"))
{
  Structure ptv = ss.Structures.First(st=>st.DicomType=="PTV");
  Segmentvolume segment = ptv.Margin(marginInMM: 5);
  Structure newStructure = ss.AddStructure("CONTROL", "PTV+5");
  newStructure.SegmentVolume = segment;
}
```

(Note that using Margin(0.0) is an easy way to copy a structure.)

With boolean functions you can modify structures further:

Code 2.28
```
Structure shell = ...
shell.SegmentVolume = newStructure.Sub(ptv);
```

Plan Generation

You can add a new course for the autoplanned plan setup like so:

Code 2.29
```
Course course;
if (patient.Courses.Where(x => x.Id == "AutoPlan").Any())
{
  course = patient.Courses.Where(x => x.Id == "AutoPlan").Single();
}
else
{
  course = curpat.AddCourse();
  course.Id = "AutoPlan";
}
```

Add a new plan setup:

Code 2.30
```
ExternalPlanSetup eps = course.AddExternalPlanSetup(ss);
eps.Id = "AutoPlanVMAT";
```

Add beams. Note that ESAPI gives options to add VMAT beams or IMRT beams, but those are meant to be used when you already have an optimized plan from your own optimizer, and you want to import that to Eclipse (for e.g. dose calculation). When you plan to run optimization, add normal arc fields or static open fields (as you would do in Eclipse UI).

Code 2.31
```
VVector isocenter = new VVector(Math.Round(ptv_high.CenterPoint.x / 10.0
    f) * 10.0f, Math.Round(ptv_high.CenterPoint.y / 10.0f) * 10.0f, Math.
    Round(ptv_high.CenterPoint.z / 10.0f) * 10.0f);

var ebmp = new ExternalBeamMachineParameters("Truebeam", "6X", 600, "ARC
    ", null);

Beam vmat1 = eps.AddArcBeam(ebmp, new VRect<double>(-100, -100, 100,
    100), 30, 181, 179, GantryDirection.Clockwise, 0, isocenter);

Beam vmat2 = cureps.AddArcBeam(ebmp, new VRect<double>(-100, -100, 100,
    100), 330, 179, 181, GantryDirection.CounterClockwise, 0, isocenter);
```

or

Code 2.32

```
Beam imrt1 = eps.AddStaticBeam(ebmp, new VRect<double>(-100, -100, 100,
    100), 0, 90, 0, isocenter);
```

Fit the collimator to the target and set calculation model and dose prescription:

Code 2.33

```
vmat1.FitCollimatorToStructure(new FitToStructureMargins(10), ptv_low,
    true, true, false);

eps.SetCalculationModel(CalculationType.PhotonVMATOptimization, "
    PO_15014");

eps.SetPrescription(NFractions, new DoseValue(2, "Gy"), 1);
```

DVH Estimation

You can create optimization objectives for the plan easily using RapidPlan. If you have suitable RapidPlan models available in the system, use the method CalculateDVHEstimates() to run estimation. Two dictionaries need to be set up before the function call. These are the same steps you normally do in the DVH Estimation user interface: map dose levels to the target structures, and match plan structure IDs to the ones used in the RapidPlan model.

Code 2.34

```
Dictionary<string, DoseValue> levels = new Dictionary<string,DoseValue
    >();
levels.Add(ptv50.Id, new DoseValue(50, "Gy"));
levels.Add(ptv70.Id, new DoseValue(70, "Gy"));
Dictionary<string, string> matches = new Dictionary<string,string>();
matches.Add(ptv70.id, "PTV_High");
matches.Add(ptv50.id, "PTV_Low");
eps.SetCalculationModel(CalculationType.DVHEstimation,
    DVHEstimationAlgorithm);
eps.CalculateDVHEstimates(modelId: "Prostate", targetDoseLevels: levels,
     structureMatches: matches);
```

After a successful calculation, the optimization objectives are automatically added to the plan. The estimate curves for upper and lower bound can be found in ExternalPlanSetup.DVHEstimates property.

Optimization

If you haven't used RapidArc to create optimization objectives automatically, you must set DVH objectives before starting the optimization.

ExternalPlanSetup has a property OptimizationSetup that gives you access to objectives:

Code 2.35
```
Structure ptvLowOpt = ...
eps.OptimizationSetup.AddPointObjective(ptvLowOpt,
    OptimizationObjectiveOperator.Lower, new DoseValue(
    doseobjectivevalue_low, "Gy"), 100, 100);

eps.OptimizationSetup.AddPointObjective(ptvLowOpt,
    OptimizationObjectiveOperator.Upper, new DoseValue(
    doseobjectivevalue_low+3.0f, "Gy"), 30, 50);

eps.OptimizationSetup.AddNormalTissueObjective(80.0f, 0.0f, 100.0f, 40.0
    f, 0.05f);
```

Already existing objectives can be found in ExternalPlanSetup.OptimizationSetup.Objectives. Iterate through them like this:

Code 2.36
```
foreach (var objective in eps.OptimizationSetup.Objectives.OfType<
    OptimizationPointObjective>())
{
  ...
}
```

And similarly for other types of objectives. If you don't use the OfType extension (or cast the type in some other way), you get a list of objects of a base type that does not show all the properties that the object has.

There is also ExternalPlanSetup.OptimizationSetup.Parameter for fluence smoothing etc.

After setting the objectives you can call the optimization function:

Code 2.37
```
eps.SetCalculationModel(CalculationType.PhotonVMATOptimization,
    OptimizationAlgorithm);
eps.OptimizeVMAT();
```

or

Code 2.38
```
eps.SetCalculationModel(CalculationType.PhotonIMRTOptimization,
    OptimizationAlgorithm);
eps.Optimize();
```

Optimize() has also variants where you can define the maximum number of iterations, continue optimization, or intermediate dose.

Leaf sequencing

Leaf sequencing for IMRT fields is run by calling

Code 2.39
```
eps.CalculateLeafMotions();
```

This version uses the LMC options that are found in the plan (most often the default LMC options). If you want to overrun those, you can use other versions of the method, for example:

Code 2.40
```
eps.SetCalculationModel(CalculationType.PhotonLeafMotions,
    LeafMotionCalculator);
eps.CalculateLeafMotions(new SmartLMCOptions(true, false));
```

This defines that the algorithm to be used is SmartLMC, and the field borders should be fixed, and jaw tracking not used. Other options you can use are LMCVOptions and LMCMSSOptions.

MCO

After optimaztion, the trade-off exploration context of PlanSetup (eps.TradeoffExplorationContext) can be used for Multicriteria Optimization (MCO).

Code 2.41
```
//Retrieve trade-off canfidates
var tradeoffCandidate = eps.TradeoffExplorationContext.
    TradeoffStructureCandidates;
...
//Add sturctures into trade-off objectives
eps.TradeoffExplorationContext.AddTradeoffObjective(sturcture);
...
//Create plan collection
if (eps.TradeoffExplorationContext.CanCreatePlanCollection)
{
        eps.TradeoffExplorationContext.CreatePlanCollection(false,
            TradeoffPlanGenerationIntermediateDoseMode.NotUsed);
}
//Navigate with the trade-off objectives
var cost = eps.TradeoffExplorationContext.GetObjectiveCost(
    tradeoffObjective);
var lowerLimit = eps.TradeoffExplorationContext.GetObjectiveLowerLimit(
    tradeoffObjective);
var upperLimit = eps.TradeoffExplorationContext.GetObjectiveUpperLimit(
    tradeoffObjective);
var newRestrictorPos = cost + (upperLimit - lowerLimit) * 0.5;
eps.TradeoffExplorationContext.SetObjectiveUpperRestrictor(
    tradeoffObjective, newRestrictorPos);
eps.TradeoffExplorationContext.SetObjectCost(tradeoffObjective,
    costForTradeoffObjective);
...
//Save and apply
eps.TradeoffExplorationContext.ApplyTradeoffExplorationResult();
app.SaveModifications();
```

Dose Calculation

Calling the dose calculation is simple:

Code 2.42
```
eps.SetCalculationModel(CalculationType.PhotonVolumeDose,
    DoseCalculationAlgorithm);
eps.CalculateDose();
```

There is also a version of calculating dose with "Fixed MUs":

Code 2.43
```
var presetValues = new List<KeyValuePair<string, MetersetValue>>();
presetValues.Add(new KeyValuePair<string,MetersetValue>(field1.Id, new
    MetersetValue(100, DosimeterUnit.MU)));
eps.CalculateDoseWithPresetValues(presetValues);
```

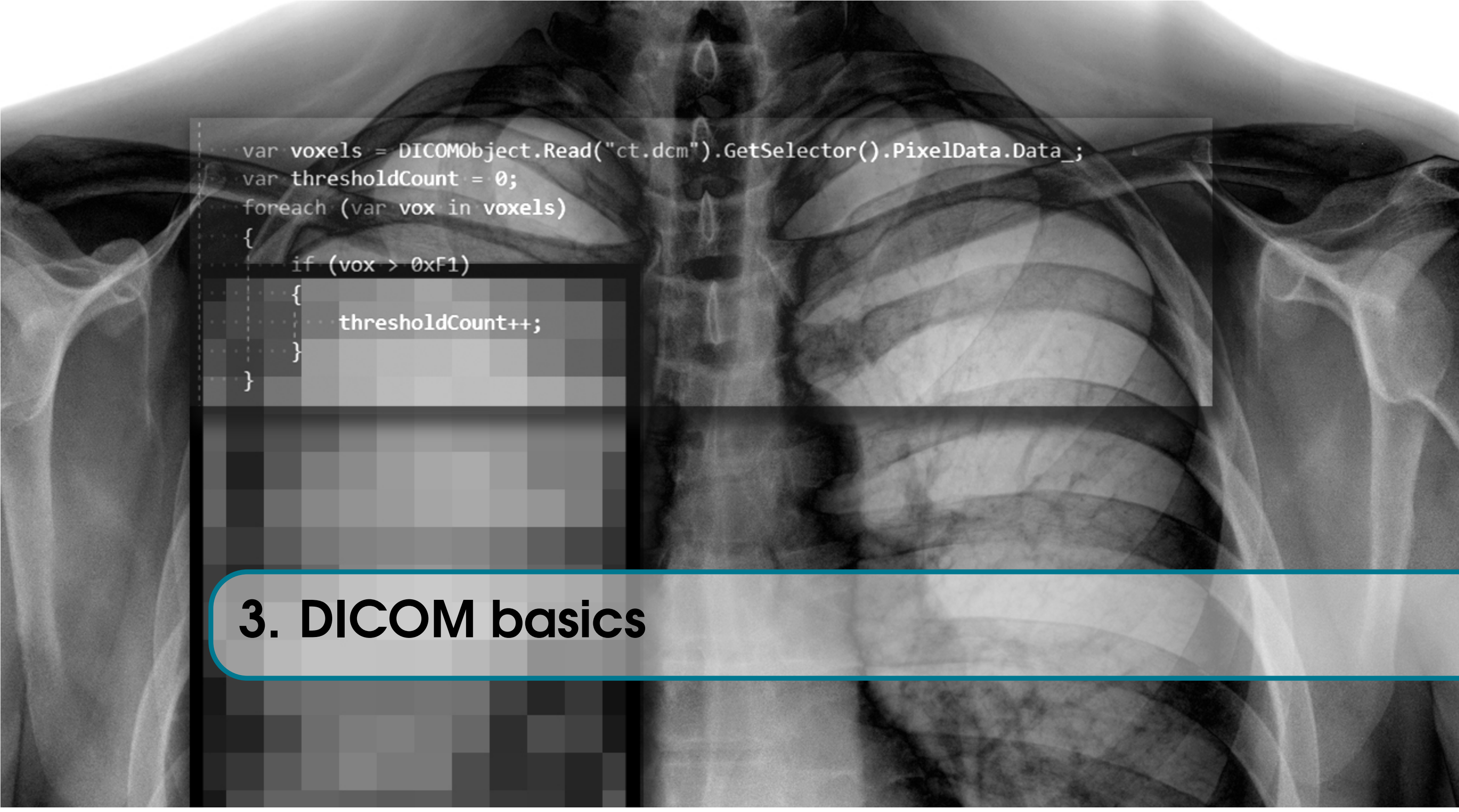
Note that this version only takes the preset values into account, when the field in an IMRT field. (For other types of fields, you can easily set the MUs by adjusting the field weight after dose calculation. The relation between field weight and field MUs is linear.)

Verification Plan Creation

When creating a verification plan, you typically want to recreate the plan on a phantom image. The first step is to copy the image from a phantom patient to the current patient. After that you use a special method to create the verification plan.

The first 'null' in the sample below is a place where you can put target study, if you have one already that you want to use. The second 'null' is a place where you can specify the study id under which the image can be found. If the image id itself is unique under the phantom patient, the study id is not needed.

Code 2.44
```
string error;
if (pat.CanCopyImageFromOtherPatient(null, "PhantomPatient", null, "CCI-
    image", out error))
{
var phantomStructureSet = pat.CopyImageFromOtherPatient("UITScript-
    Breast", null, "CCI-image");
var verificationPlan = context.Course.
    AddExternalPlanSetupAsVerificationPlan(phantomStructureSet, eps);
}
```



```
var voxels = DICOMObject.Read("ct.dcm").GetSelector().PixelData.Data_;
var thresholdCount = 0;
foreach (var vox in voxels)
{
    if (vox > 0xF1)
    {
        thresholdCount++;
    }
}
```

3. DICOM basics

REX CARDAN, PHD

3.1 Introduction

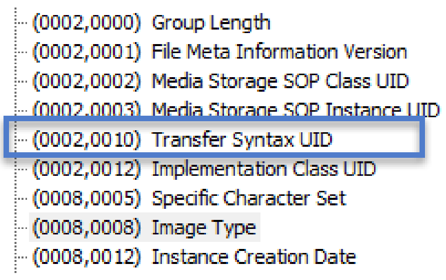
Digital Imaging and Communications in Medicine (DICOM) is the fundamental data storage and communication method in medical applications. In radiation oncology, CT images are taken and sent (via DICOM transport) to the planning system. DICOM structure sets encapsulate the CT set with segmented voxel information. A DICOM plan is created and sent to an accelerator which can process the file and deliver to a patient. DICOM RT images are acquired to make sure the patient is aligned and DICOM treatment records are stored to ensure the patient receives the correct radiation dose. In summary, DICOM is everywhere in our field and scripters need to be very familiar with its application and its design.

3.1.1 DICOM Structure

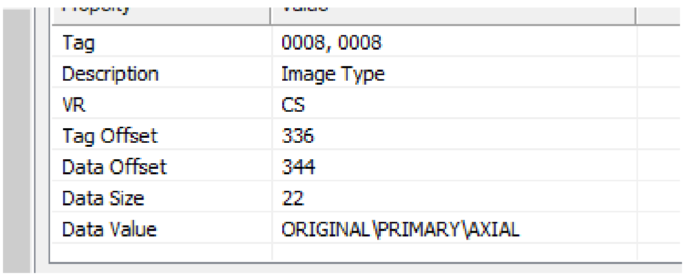
A DICOM file is a custom designed binary format. The technical specifications of the format are listed in a long (and riveting) DICOM standard set of documents (https://www.dicomstandard.org/current/). The basic idea is there is an outermost DICOM object which contains snippets of information called DICOM elements. The DICOM element is the fundamental building block of DICOM objects. Each element contains 3 main parts : an ID called a tag, a data type called a value representation (VR), and the actual data.

3.1.2 DICOM Tag

The ID (Tag) is a hexadecimal identifier for each element. Each hexadecimal is two characters long (00,01,..FF). The first two hexadecimal values represent the group ID and the last two values are the element ID within the group. For example, in the following image you can see the elements of a DICOM CT slice. In particular, I have highlighted the element Transfer Syntax UID (0002, 0010) with the group ID 0x00,0x02 and the element ID 0x00,0x10.



```
(0002,0000)  Group Length
(0002,0001)  File Meta Information Version
(0002,0002)  Media Storage SOP Class UID
(0002,0003)  Media Storage SOP Instance UID
(0002,0010)  Transfer Syntax UID
(0002,0012)  Implementation Class UID
(0008,0005)  Specific Character Set
(0008,0008)  Image Type
(0008,0012)  Instance Creation Date
```



Tag	0008, 0008
Description	Image Type
VR	CS
Tag Offset	336
Data Offset	344
Data Size	22
Data Value	ORIGINAL\PRIMARY\AXIAL

3.1.3 Value Representation

The value representation or VR is the data type for the element. It is synonymous with string, int, long, etc. in programming languages but with a richer variety. The full list of VR types supported in the newest DICOM Standard are listed below. One of the VRs is the sequence type which allows nesting of DICOM elements. It is similar to a List<T> or Array type in .NET. When a DICOM object contains sequence elements, the structure resembles a tree structure, a list of lists, like XML. VR types are either explicitly set in the element (called Explicit Little/Big Endian syntax) or they are known implicitly by the element ID. For example, the DICOM element Transfer Syntax UID is always of VR type Unique Identifier. The program or library used to read DICOM files must contain a DICOM dictionary to be able to look up VRs based on the elements DICOM Tag if it is not explicitly provided.

Available VR types in DICOM Standard 2017c. From Table 6.2-1 in The DICOM Standard Part 5

- Application Entity
- Age String
- Attribute Tag
- Code String
- Date
- Decimal String
- Date Time
- Floating Point Single
- Floating Point Double
- Integer String
- Long String
- Long Text
- Other Byte
- Other Double
- Other Float
- Other Word
- Person Name
- Short String
- Signed Long
- Sequence Long
- Signed Short
- Short Text
- Time
- Unlimited Characters
- Unique Identifier
- Unsigned Long
- Unknown

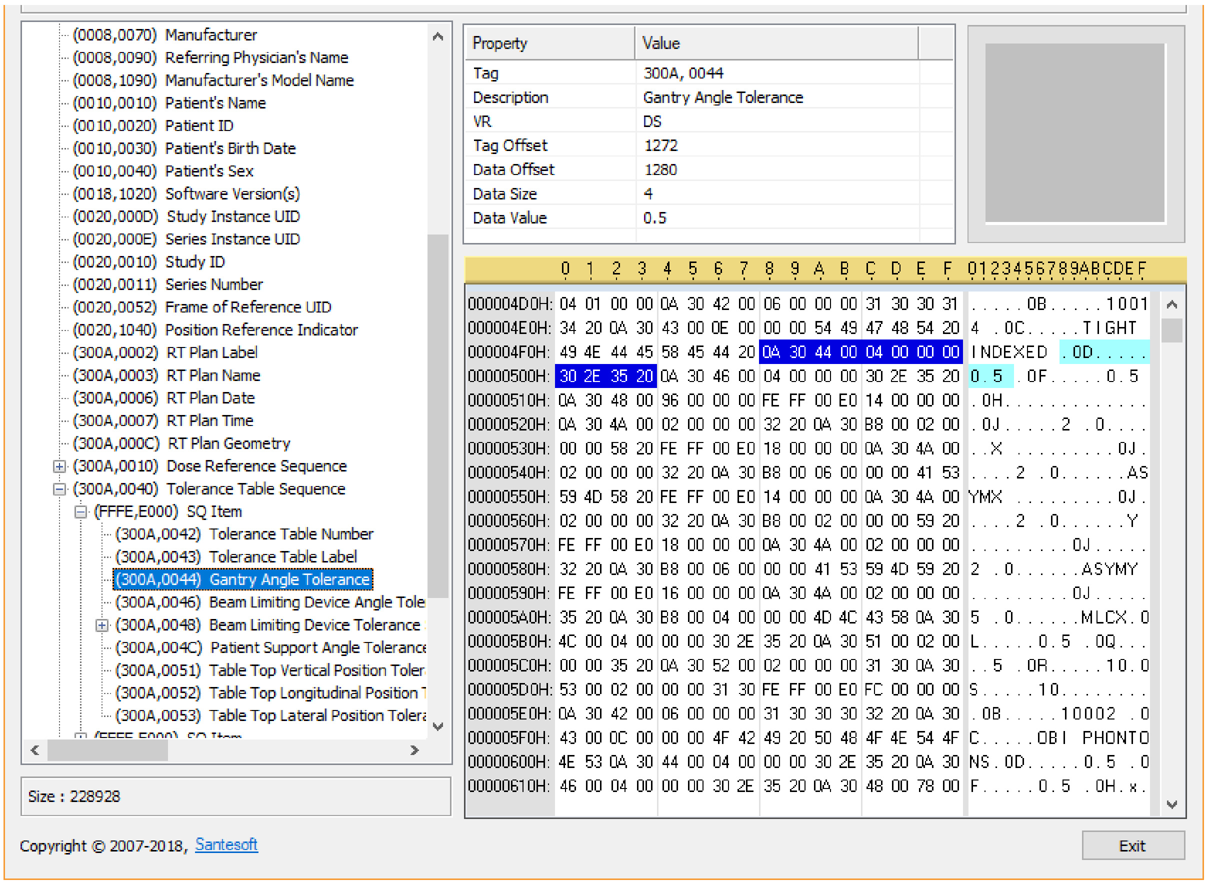
- Universal Resource Identifier
- Unsigned Short
- Unlimited Text

3.1.4 DICOM Data

The real meat of each DICOM element is of course the data. The data portion of the element must be decoded based on the VR. If the VR is Code String for example, the data must be decoded as a string. The data section can actually hold multiple values of the VR type. In this sense, almost every element is like an array. Most elements just hold one datum, but if there are multiple values (called value multiplicity greater than one), then the data will be separated by the '/' character.

3.2 Exploring DICOM Visually

If you would like to get a understanding of the layout of a DICOM object, the best thing to do is open up the file in DICOM tool. My personal favorite and lightweight application is called "Sante Hexidecimal Viewer". This viewer provides element inspection with a tree view to see the positions of various DICOM elements within the file. Additionally, it allows inspection of the actual bytes which can aid in debugging DICOM problems.



```
(0008,0070) Manufacturer
(0008,0090) Referring Physician's Name
(0008,1090) Manufacturer's Model Name
(0010,0010) Patient's Name
(0010,0020) Patient ID
(0010,0030) Patient's Birth Date
(0010,0040) Patient's Sex
(0018,1020) Software Version(s)
(0020,000D) Study Instance UID
(0020,000E) Series Instance UID
(0020,0010) Study ID
(0020,0011) Series Number
(0020,0052) Frame of Reference UID
(0020,1040) Position Reference Indicator
(300A,0002) RT Plan Label
(300A,0003) RT Plan Name
(300A,0006) RT Plan Date
(300A,0007) RT Plan Time
(300A,000C) RT Plan Geometry
(300A,0010) Dose Reference Sequence
(300A,0040) Tolerance Table Sequence
  (FFFE,E000) SQ Item
    (300A,0042) Tolerance Table Number
    (300A,0043) Tolerance Table Label
    (300A,0044) Gantry Angle Tolerance
    (300A,0046) Beam Limiting Device Angle Tole
    (300A,0048) Beam Limiting Device Tolerance
    (300A,004C) Patient Support Angle Toleranc
    (300A,0051) Table Top Vertical Position Toler
    (300A,0052) Table Top Longitudinal Position T
    (300A,0053) Table Top Lateral Position Toler

Size : 228928
```

Property	Value
Tag	300A, 0044
Description	Gantry Angle Tolerance
VR	DS
Tag Offset	1272
Data Offset	1280
Data Size	4
Data Value	0.5

```
           0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  0123456789ABCDEF
000004D0H: 04 01 00 00 0A 30 42 00 06 00 00 00 31 30 30 31  ......0B....1001
000004E0H: 34 20 0A 30 43 00 0E 00 00 00 54 49 47 48 54 20  4 .0C.....TIGHT
000004F0H: 49 4E 44 45 58 45 44 20 0A 30 44 00 04 00 00 00  INDEXED .0D.....
00000500H: 30 2E 35 20 0A 30 46 00 04 00 00 00 30 2E 35 20  0.5 .0F.....0.5
00000510H: 0A 30 48 00 96 00 00 00 FE FF 00 E0 14 00 00 00  .0H.............
00000520H: 0A 30 4A 00 02 00 00 00 32 20 0A 30 B8 00 02 00  .0J.....2 .0....
00000530H: 00 00 58 20 FE FF 00 E0 18 00 00 00 0A 30 4A 00  ..X .........0J.
00000540H: 02 00 00 00 32 20 0A 30 B8 00 06 00 00 00 41 53  ....2 .0......AS
00000550H: 59 4D 58 20 FE FF 00 E0 14 00 00 00 0A 30 4A 00  YMX .........0J.
00000560H: 02 00 00 00 32 20 0A 30 B8 00 02 00 00 00 59 20  ....2 .0......Y
00000570H: FE FF 00 E0 18 00 00 00 0A 30 4A 00 02 00 00 00  .........0J.....
00000580H: 32 20 0A 30 B8 00 06 00 00 00 41 53 59 4D 59 20  2 .0......ASYMY
00000590H: FE FF 00 E0 16 00 00 00 0A 30 4A 00 02 00 00 00  .........0J.....
000005A0H: 35 20 0A 30 B8 00 04 00 00 00 4D 4C 43 58 0A 30  5 .0......MLCX.0
000005B0H: 4C 00 04 00 00 00 30 2E 35 20 0A 30 51 00 02 00  L.....0.5 .0Q...
000005C0H: 00 00 35 20 0A 30 52 00 02 00 00 00 31 30 0A 30  ..5 .0R.....10.0
000005D0H: 53 00 02 00 00 00 31 30 FE FF 00 E0 FC 00 00 00  S.....10........
000005E0H: 0A 30 42 00 06 00 00 00 31 30 30 30 32 20 0A 30  .0B.....10002 .0
000005F0H: 43 00 0C 00 00 00 4F 42 49 20 50 48 4F 4E 54 4F  C.....OBI PHONTO
00000600H: 4E 53 0A 30 44 00 04 00 00 00 30 2E 35 20 0A 30  NS.0D.....0.5 .0
00000610H: 46 00 04 00 00 00 30 2E 35 20 0A 30 48 00 78 00  F.....0.5 .0H.x.
```

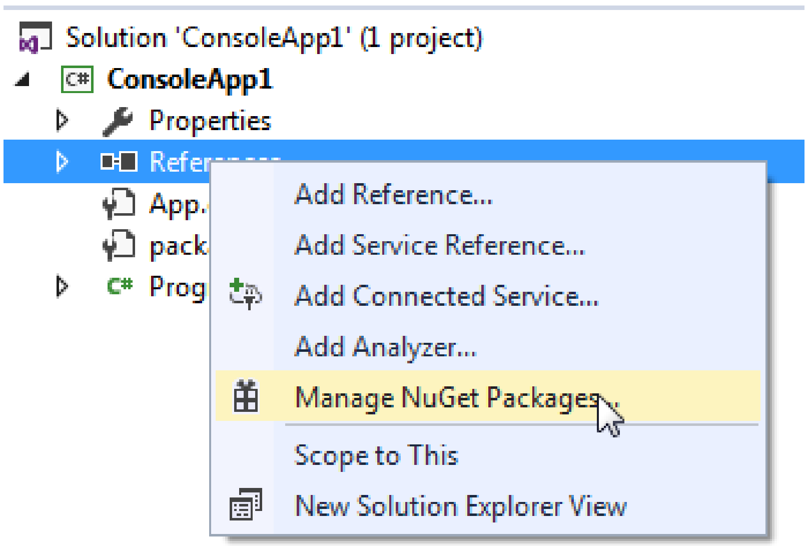
Copyright © 2007-2018, Santesoft

3.3 Exploring DICOM With Evil DICOM

Because DICOM files are specially formatted binary files, you won't be able to just open one in your favorite text editor. Instead to inspect and manipulate them, you will need to grab a DICOM library. There are some great libraries including PyDICOM, fo-dicom, dcmtk, and my personal favorite, Evil DICOM. Since I am most familiar with Evil DICOM, the following examples will use this .NET library to demonstrate the principles needed when programming against DICOM files.

3.3.1 Installing EvilDICOM via NuGet

1. Create a new C Sharp .NET console project. 2. Right click your project and choose Manage NuGet Packages 3. Search for "EvilDICOM" and choose install to automatically pull the library from the internet



3.3.2 Opening Your First DICOM File

If you don't have any DICOM files available, export some from Eclipse to your desktop. DICOM files typically end with ".dcm" extension and begin with letters which signify their file type (Varian convention). The most common file types are :

- "RP..." - RT Plan object
- "RD" - RT Dose object
- "RS" - RT Structure Set object
- "RI" - RT Image Object (DRR, kV, Portal image)
- "CT" - Computed Tomography Image Slice
- "PT" - Positron Emission Tomography Image Slice
- "MR" - Magnetic Resonance Image Slice
- "REG" - Registration Object

Choose a file that you want to inspect and copy the path. In Windows, to copy the path, hold shift when right clicking the file and choose "Copy as Path" option. To load the DICOM object into memory so you can inspect it, use the following code:

Code 3.1 — Read DICOM object from file.

```
//Your copied path below
var path = @"\\vms-image\va_data$\DICOM\CT.1.dcm";
var ctSlice = DICOMObject.Read(path);
```

A CT image slice object has many different elements which define the image. It contains the mA, kVp and relevant equipment parameters used to acquire the scan. It also contains the current window/level, position in 3D space and all of the voxels in the image. To see all of the elements, you can access the **Elements** property.

Code 3.2

```
var elements = ctSlice.Elements;
foreach(var el in elements)
 {
       Console.WriteLine($"Tag={el.Tag}");
       Console.WriteLine($"VR={el.VR}");
       Console.WriteLine($"Data={el.DData}");
}
```

■

3.3.3 Selecting Elements And Accessing Data

The DICOM elements can be selected by tag ID in most libraries. The DICOM element tag is a unique set of 4 bytes which identifies the element and gives its data meaning. The first 2 bytes are the DICOM group ID, and the last 2 bytes are the element ID. The bytes are typically shown in hexadecimal notation. For example, the DICOM element tag which holds the patient ID is (0010,0020), where the group ID is 0x00,0x10 and the element ID is 0x00,0x20. To select the patient ID in our DICOM object, you would do the following:

Code 3.3

```
var patientIdElement = ctSlice.FindFirst("00100020");
var patientId = (string)patientIdElement.DData;
```

■

All public DICOM element tags are defined in Part 6 of the DICOM standard (http://dicom.nema.org/medical/dicom/curr
You will need to spend the next few days memorizing all of the thousands of the DICOM tags. Go ahead and pause your progress in the book and come back when you are finished. Still reading? Of course, for those who are lazy (hint: you), there are a lot of helper methods to make it easy to select elements without knowing the tag.

3.3.4 The Tag Helper

The first way you can access tag IDs is to use the **TagHelper** class found in *EvilDICOM.Core.Helpers* namespace. The tag helper just finds the ID using static strings of the name of the element. For example, to find the Patient Id element you can write:

Code 3.4

```
var patientIdElement = ctSlice.FindFirst(TagHelper.PatientID)
    ;
 var patientId = (string)patientIdElement.DData;
```

■

3.3.5 The Selector Class

The next method of selection is my personal favorite. If you look at the previous example, you finding an element through an interface of IDICOMElement which has properties called **DData** and **DData_**. These are just holders for data but the type is unknown. Because of this, a cast must be performed. Evil DICOM has a neat way to strongly type the data so you can take advantage of the compiler checks in Visual Studio. The magic trick is the class called DICOMSelector. The DICOMSelector class is like the TagHelper class in that it has properties for each DICOM element in part 6 of the standard. But unlike the selection we did before, the elements selected by the DICOMSelector class contain properties **Data** and **Data_** which are strongly typed. To select with this method, you call the GetSelector() method on a DICOM object as follows:

Code 3.5
```
var selector = ctSlice.GetSelector();
 var patientID = selector.PatientID.Data;
```

3.3.6 DICOM Data

DICOM elements can hold one or more datum of the type specified by the VR. If you are wanting to inspect just the first datum or if you believe only one datum value is present, you use the **Data** property as we saw in the above example. However, if you want to inspect or manipulate multiple values, you must use the **Data_** (with a trailing underscore) in Evil DICOM. For example, the position of a CT image slice is stored in element (0020,0032) - Image Position. The data is an array of 3 values : X, Y, and Z location. You can access these values in the following way:

Code 3.6
```
// Notice underscore
 var position = selector.ImagePosition.Data_;
 var x = position[0];
 var y = position[1];
 var z = position[2];
```

3.3.7 Working With Sequences

Sequences are elements which contain more elements. However, a sequence's children are not DICOM elements. They are Sequence Items. A sequence item can be thought of as another DICOM object in that it is just a collection of elements. So a sequence then is just a container for descendant DICOM objects. This hierarchy can be seen in the diagram below. Sequences are very important in

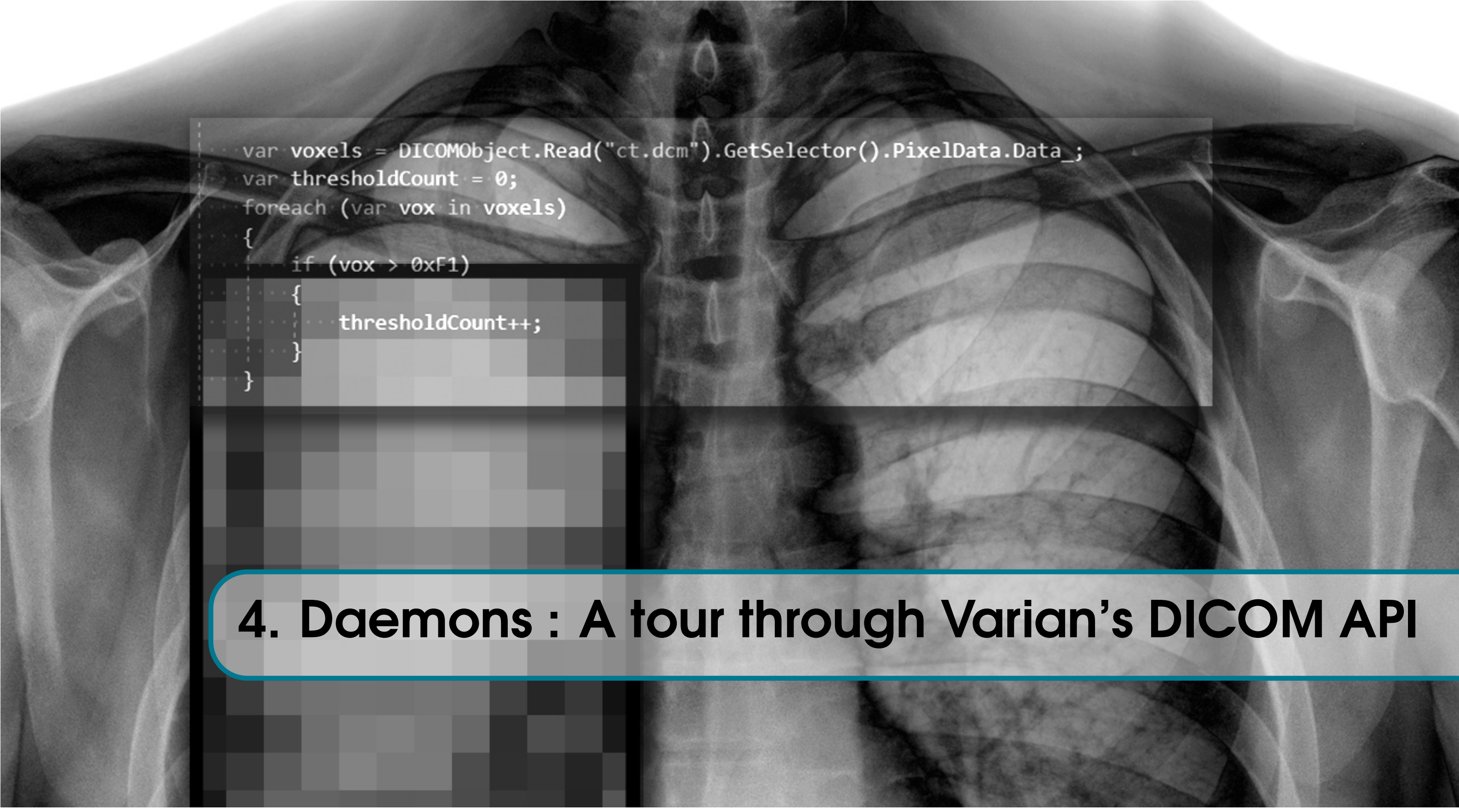
3.3.8 Hacking DICOM Files

So far we have only been concerned with reading DICOM files, but you can also change data and write new *hacked* files. For example, if you wanted to changed the patient name and ID, you could execute:

Code 3.7
```
var selector = ctSlice.GetSelector();
selector.PatientName.Data = "FLINSTONE^FRED";
selector.PatientID.Data = "123456";
ctSlice.Write(path);
```

3.4 Conclusion

This chapter has been a review of the DICOM structure and some tools to get started programming against DICOM files. Having established a foundational understanding of the DICOM file format, we can extend our journey into the more complex networking operations.



```
var voxels = DICOMObject.Read("ct.dcm").GetSelector().PixelData.Data_;
var thresholdCount = 0;
foreach (var vox in voxels)
{
    if (vox > 0xF1)
    {
        thresholdCount++;
    }
}
```

4. Daemons : A tour through Varian's DICOM API

REX CARDAN, PHD

4.1 Introduction

DICOM network operations are the backbone of a radiation oncology operation. Files are searched, moved, stored all in an advanced protocol defined in Part 7 of the DICOM standard (https://www.dicomstandard.org/current/). The network nodes which are sending and receiving DICOM messages are often referred to as DICOM daemons (pronounced demons). The communication is a special DICOM specific language layered on TCP/IP protocol. In this chapter, we will learn the language of the daemons allowing for some very advanced clinical and research operations.

DICOM Service Classes

Typically DICOM nodes on the network are referred to as service class users (SCU) or service class providers (SCP). SCUs are the clients who are usually initiating calls and the SCPs are responding to the calls and sending back the data requested. All DICOM services have an identity on the network consisting of 3 main properties: IP address, communication port, and application entity title (AE title). These unique properties allow for secure transmission of data on a network. Most commercial SCPs will have a white-list of services to which they are allowed to communicate. Any request coming from entities not listed in the white-list is considered hostile and calls will not be answered.

Source Code

The source code from this chapter is available on Github in the VarianAPIs repositories. The web address is `https://github.com/VarianAPIs/DICOM_Communication_101`. The solution file contains each example with a program class that runs the requested tutorial. Site specific details will have to be modified (IP addresses, etc) for it to work properly.

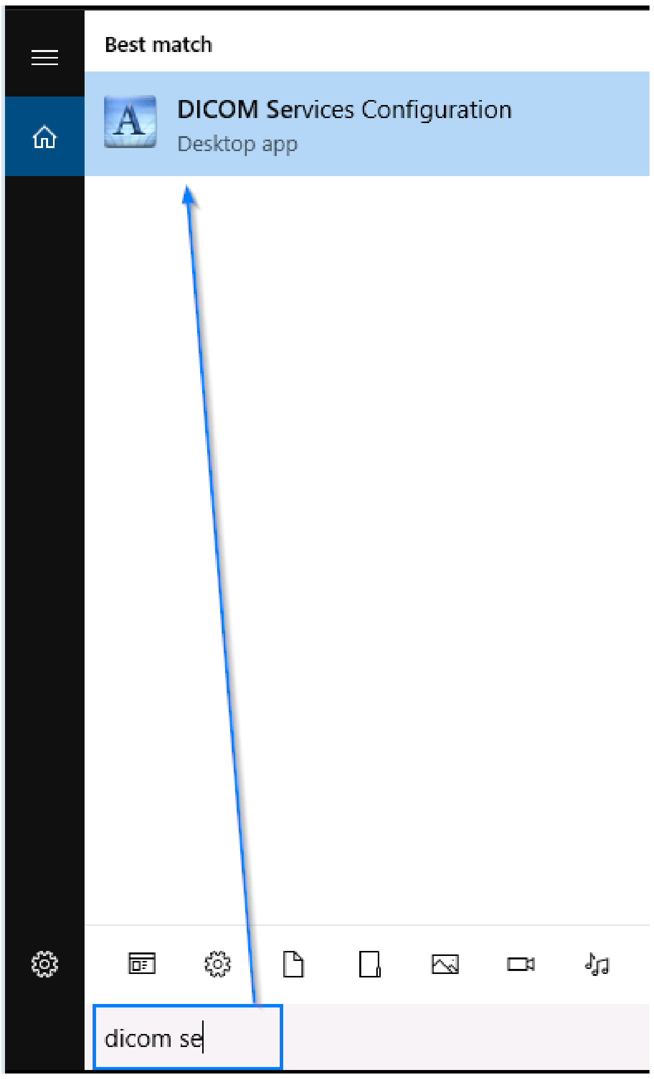
4.2 Setting Up a Varian Daemon

Before we get to exploring the communication details, we need to set up our Varian DICOM environment. The provided SCP (called the daemon from here on) is a software installation separate from Aria and Eclipse. In versions prior to Aria 15, the DICOM daemon could be installed on any Eclipse thick client workstation. In versions 15 and higher, the DICOM services all run from a single server. This tutorial will cover the setup principles of the DICOM daemon which will work with all Aria versions. However the images and exact steps will be taken from the setup in version 15.5.

4.2.1 Finding the DICOM Services Server

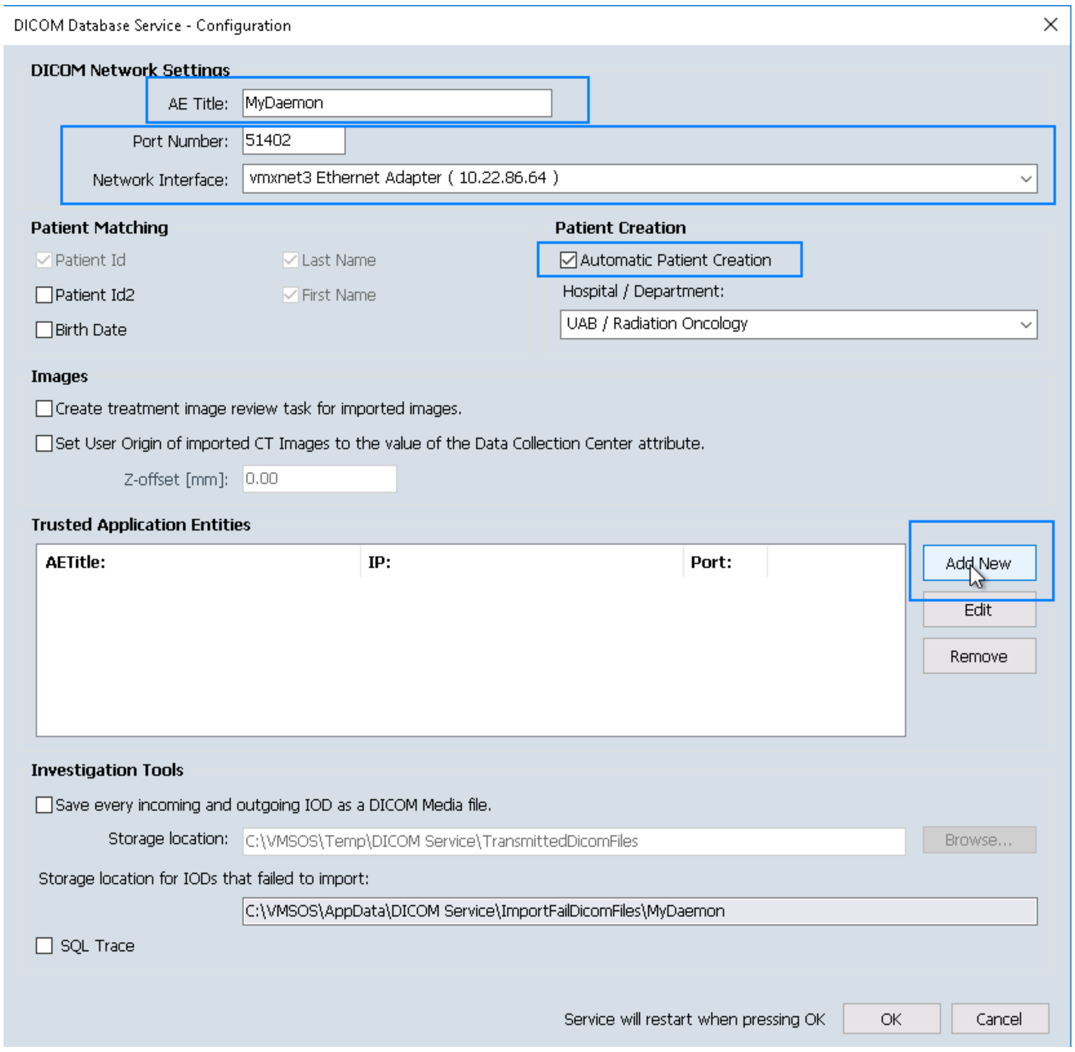
You will need administrative rights to be able to proceed through this tutorial, and mis-configuring or changing existing DICOM services can be very bad. The DICOM services server can be found from the Varian Service Portal (web service). From there, go to **Inventory » Workstation Details**. There you will find a list of all Varian servers and workstations on your network. Search for the one that has the "DCM" in the name. This is the server you will remote desktop (RDP) into in order to configure the daemon. One you RDP into the DICOM services server:

1. Type **Windows + Q** to open the Windows search
2. Enter the text "**DICOM ser...**"
3. Select **DICOM Services Configuration**



4.2.2 Spawning a Daemon

1. Select **Add Service**
2. Select **Database Service**
3. Set the **AE Title**, **port number**, and record the **IP address** from the ethernet adapter.
4. **Check** Automatic Patient Creation
5. Select **Add New** (trusted entity)


DICOM Database Service - Configuration

DICOM Network Settings

AE Title: MyDaemon

Port Number: 51402

Network Interface: vmxnet3 Ethernet Adapter (10.22.86.64)

Patient Matching

- [✓] Patient Id
- [✓] Last Name
- [ ] Patient Id2
- [✓] First Name
- [ ] Birth Date

Patient Creation

- [✓] Automatic Patient Creation

Hospital / Department: UAB / Radiation Oncology

Images

- [ ] Create treatment image review task for imported images.
- [ ] Set User Origin of imported CT Images to the value of the Data Collection Center attribute.

Z-offset [mm]: 0.00

Trusted Application Entities

AETitle:	IP:	Port:

Investigation Tools

- [ ] Save every incoming and outgoing IOD as a DICOM Media file.

Storage location: C:\VMSOS\Temp\DICOM Service\TransmittedDicomFiles

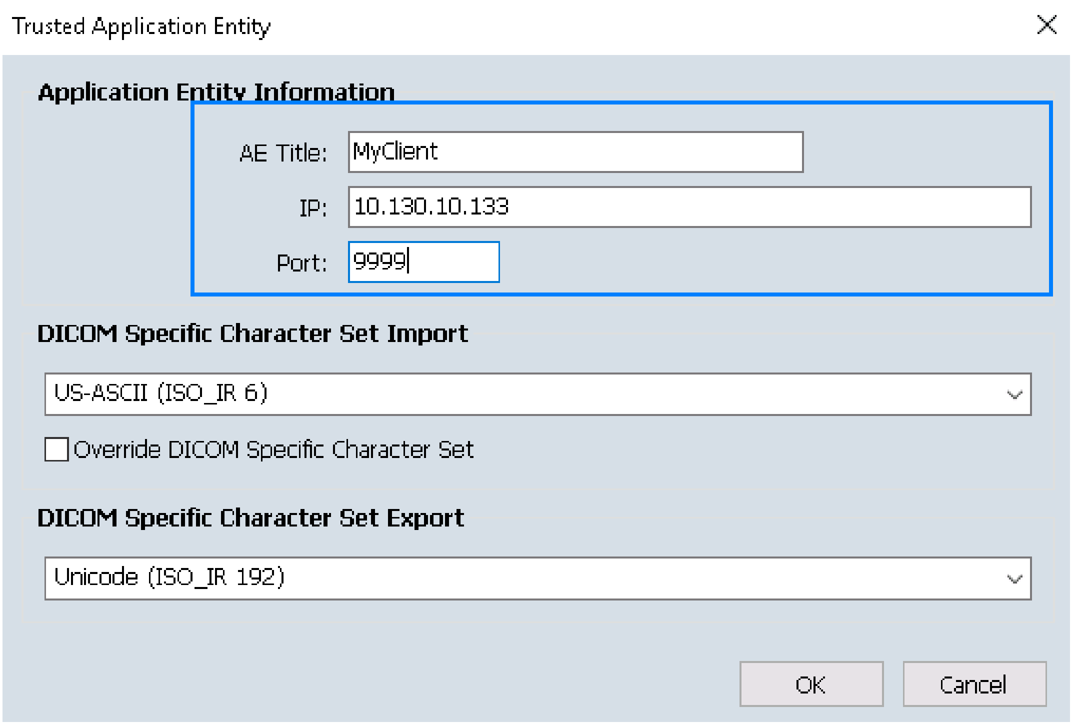
Storage location for IODs that failed to import: C:\VMSOS\AppData\DICOM Service\ImportFailDicomFiles\MyDaemon

- [ ] SQL Trace

Service will restart when pressing OK

4.2.3 Adding a Trusted Entity

This next step is site specific. You will need to whitelist some computers so that they can talk to the daemon. For each computer that will need to communicate with the daemon, you will have to add a new trusted entity. The AE title just needs to be a unique name for the DICOM nodes. The IP address will be the address for the computer from which you will be calling (probably where Visual Studio is installed). Remember the port that you assign here. It will be important for some operations.


Trusted Application Entity

Application Entity Information

AE Title: MyClient

IP: 10.130.10.133

Port: 9999

DICOM Specific Character Set Import

US-ASCII (ISO_IR 6)

☐ Override DICOM Specific Character Set

DICOM Specific Character Set Export

Unicode (ISO_IR 192)

We are now ready to start communicating. The Varian daemon can perform several useful operations for a client. Each of the operations is explored in the following sections.

4.3 DICOM Language Basics

The exact bitwise operations of the protocol are beyond the scope of this chapter. Instead we will focus on high level communication concepts more relevant to the DICOM library user. If instead, you were trying to develop a library or implement a DICOM call from scratch, I recommend the book Digital Imaging and Communications in Medicine (DICOM): A Practical Introduction and Survival Guide by Pianykh. Additionally, the DICOM transport layer documentation (part 8) provides very detailed information. For the purposes of learning DICOM communication using an existing library, we will continue to use Evil DICOM. For details on setting up a project with Evil DICOM, see the previous chapter. The main DICOM calls we are interested in here are the following:

- C-ECHO (DICOM "ping")
- C-FIND (DICOM search)
- C-MOVE (DICOM move and storage)
- C-STORE (DICOM storage)

4.3.1 C-ECHO

The first network operation is the simplest. When starting to set up a DICOM network, the DICOM creators designed a mechanism equivalent to the ping command in Windows. Successful execution of the following command ensures that 1) the DICOM node is up and running and 2) it is allowed to communicate with the client.

Code 4.1

```
//Store the details of the daemon (Ae Title, IP, port)
var daemon = new Entity("PHYSX_DICOM", "10.22.86.64", 51402);
//Store the details of the client (Ae Title, port) -> IP address is
    determined by CreateLocal() method
var local = Entity.CreateLocal("DICOMEC1", 9999);
//Set up a client (DICOM SCU = Service Class User)
var client = new DICOMSCU(local);
//TRY C-ECHO
var canPing = client.Ping(daemon);
//Write results to console
Console.WriteLine($"DICOM␣C-Echo␣from␣{local.AeTitle}␣=>␣" + $"{daemon.
    AeTitle}␣@{daemon.IpAddress}:{daemon.Port}␣was␣successfull?␣{canPing}
    ");

Console.Read(); //Stop here
```

4.3.2 C-FIND

The C-FIND operation allows searching patient DICOM files using certain keys such as patient Id, SOP instance UIDs, etc. Typically, unless you know SOP UIDs, you will need to start with the patient Id and work your way down from studies to series, and then from series to images. The convention in DICOM is to refer to any DICOM file as a DICOM "image". This will include RT plans, dose files, and structure sets as well as actual image data. If you know the patient Id, then you can use the following technique to find all studies, series and images.

Code 4.2

```
//Store the details of the daemon (Ae Title, IP, port)
var daemon = new Entity("PHYSX_DICOM", "10.22.86.64", 51402);
//Store the details of the client (Ae Title, port) -> IP address is
    determined by CreateLocal() method
var local = Entity.CreateLocal("DICOMEC1", 9999);
//Set up a client (DICOM SCU = Service Class User)
var client = new DICOMSCU(local);
//Build a finder class to help with C-FIND operations
var finder = client.GetCFinder(daemon);

var studies = finder.FindStudies("DA00001");
var series = finder.FindSeries(studies);
var images = finder.FindImages(series);

//Write results to console
Console.WriteLine($"DICOM␣C-Find␣from␣{local.AeTitle}␣=>␣" +
        $"{daemon.AeTitle}␣@{daemon.IpAddress}:{daemon.Port}:");
Console.WriteLine($"{studies.Count()}␣Studies␣Found");
Console.WriteLine($"{series.Count()}␣Series␣Found");
Console.WriteLine($"{images.Count()}␣Images␣Found");
Console.Read(); //Stop here
```

4.3.3 C-MOVE

Finding DICOM files is great, but you typically want to do something with those results. Enter C-MOVE. The C-MOVE operation is used to take results from a C-FIND and send them to a DICOM node on the network (which can be the same as the DICOM node you are calling from). At UAB, we use this technique to rapidly send patient DICOM files to Mobius 3D for rapid commissioning of the system. You can also send DICOM files to a PACs or even back to Aria. The following example shows how to move to a separate destination (Mobius in this case).

Code 4.3

```
//Store the details of the daemon (Ae Title, IP, port)
var daemon = new Entity("PHYSX_DICOM", "10.22.86.64", 51402);
//Store the details of the mobius DICOM entity (Ae Title, IP, port)
var mobius = new Entity("MOBIUST", "10.241.20.41", 104);
//Store the details of the client (Ae Title, port) -> IP address is
    determined by CreateLocal() method
var local = Entity.CreateLocal("DICOMEC1", 9999);
//Set up a client (DICOM SCU = Service Class User)
var client = new DICOMSCU(local);
//Build a finder class to help with C-FIND operations
var finder = client.GetCFinder(daemon);
var studies = finder.FindStudies("DA00001");
var series = finder.FindSeries(studies);

//Filter series by modality, then create list of
var plans = series.Where(s => s.Modality == "RTPLAN")
    .SelectMany(ser => finder.FindImages(ser));
var doses = series.Where(s => s.Modality == "RTDOSE")
        .SelectMany(ser => finder.FindImages(ser));
var cts = series.Where(s => s.Modality == "CT")
        .SelectMany(ser => finder.FindImages(ser));

var mover = client.GetCMover(daemon);
ushort msgId = 1;
foreach(var plan in plans)
{
    Console.WriteLine($"Sending plan {plan.SOPInstanceUID}...");
    //Make sure Mobius is on the whitelist of the daemon
    var response = mover.SendCMove(plan, mobius.AeTitle, ref msgId);
    Console.WriteLine($"DICOM C-Move Results : ");
    Console.WriteLine($"Number of Completed Operations : {response.
        NumberOfCompletedOps}");
    Console.WriteLine($"Number of Failed Operations : {response.
        NumberOfFailedOps}");
    Console.WriteLine($"Number of Remaining Operations : {response.
        NumberOfRemainingOps}");
    Console.WriteLine($"Number of Warning Operations : {response.
        NumberOfWarningOps}");
}

Console.Read(); //Stop here
```

■

4.3.4 C-MOVE To Self

Of course a lot of times, you just want to get the DICOM files to do something with them. You can actually instruct the daemon to move to the client node. It is a little trickier to do this because you will need to create a separate "receiver" to catch the files when they come back. Because these two nodes will need to run at the same time, they will need to be run on two separate threads. Evil DICOM will help you manage the threading part if you are using that library. The following example shows how to setup a DICOM service class provider to catch the files as they get sent back to the client.

Code 4.4

```
//Set up a receiver to catch the files as they come in
var receiver = new DICOMSCP(local);
//Let the daemon know we can take anything it sends
receiver.SupportedAbstractSyntaxes = AbstractSyntax.
    ALL_RADIOTHERAPY_STORAGE;
//Set up storage location
var desktopPath = Environment.GetFolderPath(Environment.SpecialFolder.
    Desktop);
var storagePath = Path.Combine(desktopPath, "DICOM␣Storage");
Directory.CreateDirectory(storagePath);
//Set the action when a DICOM files comes in
receiver.DIMSEService.CStoreService.CStorePayloadAction = (dcm, asc) =>
{
    var path = Path.Combine(storagePath, dcm.GetSelector().
        SOPInstanceUID.Data + ".dcm");
    Console.WriteLine($"Writing␣file␣{path}...");
    dcm.Write(path);
    return true; // Lets daemom know if you successfully wrote to drive
};
receiver.ListenForIncomingAssociations(true);
```

C-MOVE To Self : Full Example

To send to yourself (the initiating client), simply change the AE title in the mover.SendCMove(..) method to the local entity title. The full example of this process can be seen in the following code,

Code 4.5

```
//Store the details of the daemon (Ae Title, IP, port)
var daemon = new Entity("PHYSX_DICOM", "10.22.86.64", 51402);
//Store the details of the client (Ae Title, port) -> IP address is
    determined by CreateLocal() method
var local = Entity.CreateLocal("DICOMEC1", 9999);
//Set up a client (DICOM SCU = Service Class User)
var client = new DICOMSCU(local);
//Set up a receiver to catch the files as they come in
var receiver = new DICOMSCP(local);
//Let the daemon know we can take anything it sends
receiver.SupportedAbstractSyntaxes = AbstractSyntax.
    ALL_RADIOTHERAPY_STORAGE;
//Set up storage location
var desktopPath = Environment.GetFolderPath(Environment.SpecialFolder.
    Desktop);
var storagePath = Path.Combine(desktopPath, "DICOM␣Storage");
Directory.CreateDirectory(storagePath);
//Set the action when a DICOM files comes in
receiver.DIMSEService.CStoreService.CStorePayloadAction = (dcm, asc) =>
{
    var path = Path.Combine(storagePath, dcm.GetSelector().
        SOPInstanceUID.Data + ".dcm");
    Console.WriteLine($"Writing␣file␣{path}...");
    dcm.Write(path);
    return true; // Lets daemom know if you successfully wrote to drive
};
receiver.ListenForIncomingAssociations(true);
```

```
//Build a finder class to help with C-FIND operations
var finder = client.GetCFinder(daemon);
var studies = finder.FindStudies("DA00001");
var series = finder.FindSeries(studies);

//Filter series by modality, then create list of
var plans = series.Where(s => s.Modality == "RTPLAN")
    .SelectMany(ser => finder.FindImages(ser));
var doses = series.Where(s => s.Modality == "RTDOSE")
        .SelectMany(ser => finder.FindImages(ser));
var cts = series.Where(s => s.Modality == "CT")
        .SelectMany(ser => finder.FindImages(ser));

var mover = client.GetCMover(daemon);
ushort msgId = 1;
foreach (var plan in plans)
{
    Console.WriteLine($"Sending␣plan␣{plan.SOPInstanceUID}...");
    //Make sure Mobius is on the whitelist of the daemon
    var response = mover.SendCMove(plan, local.AeTitle, ref msgId);
    Console.WriteLine($"DICOM␣C-Move␣Results␣:␣");
    Console.WriteLine($"Number␣of␣Completed␣Operations␣:␣{response.
       NumberOfCompletedOps}");
    Console.WriteLine($"Number␣of␣Failed␣Operations␣:␣{response.
       NumberOfFailedOps}");
    Console.WriteLine($"Number␣of␣Remaining␣Operations␣:␣{response.
       NumberOfRemainingOps}");
    Console.WriteLine($"Number␣of␣Warning␣Operations␣:␣{response.
       NumberOfWarningOps}");
}

Console.Read(); //Stop here
```

4.3.5 C-STORE

All of the previous operations have relied on the daemon to generate the DICOM files as we need them. What about if we already have DICOM files and we want to send them for storage to Aria? The last operation we will review, called the C-Store operation is used in just this case. To send files to the daemon, you can use the following method.

Code 4.6

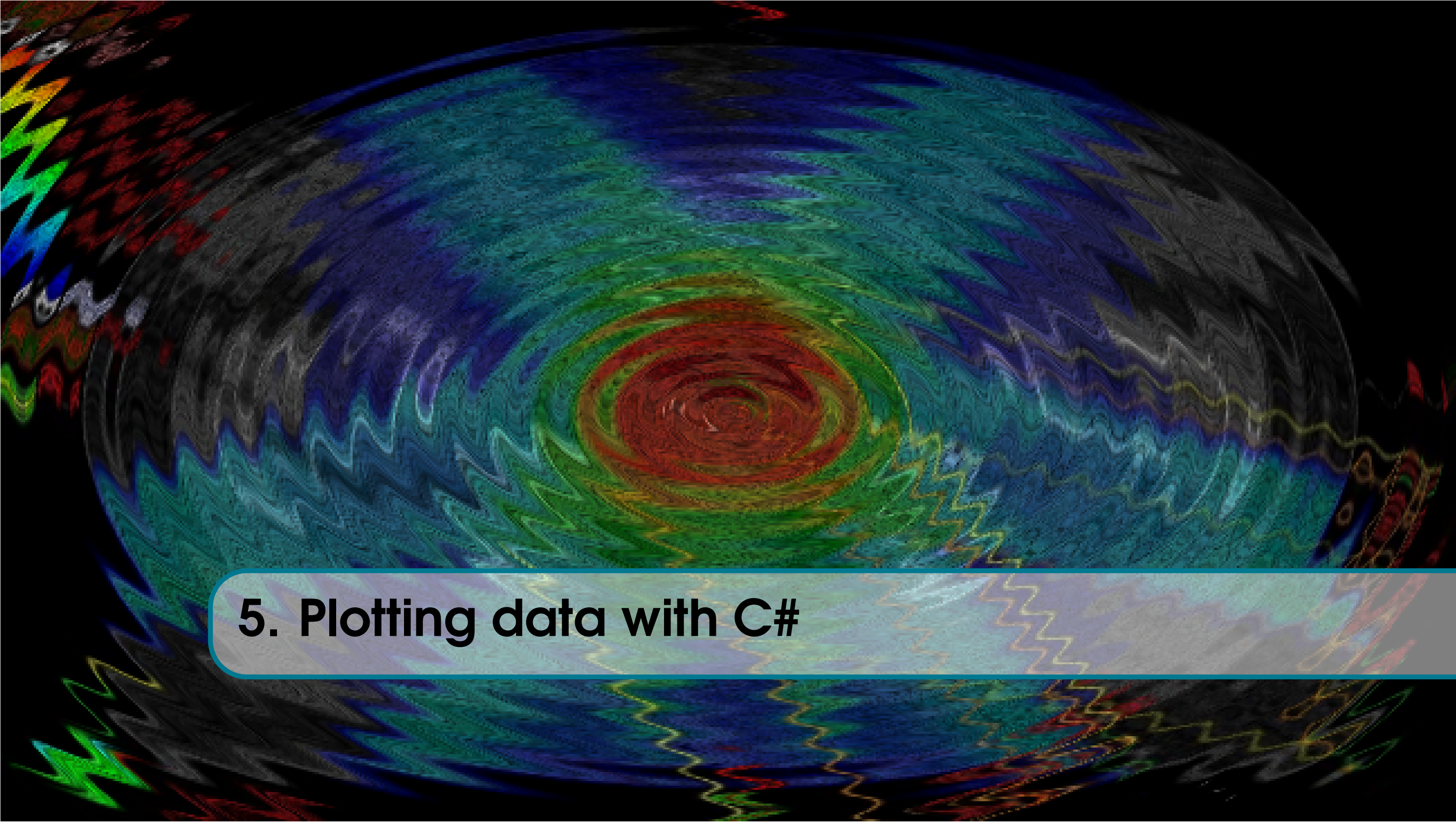
```
//Store the details of the daemon (Ae Title, IP, port)
var daemon = new Entity("PHYSX_DICOM", "10.22.86.64", 51402);
//Store the details of the client (Ae Title, port) -> IP address is
    determined by CreateLocal() method
var local = Entity.CreateLocal("DICOMEC1", 9999);
//Set up a client (DICOM SCU = Service Class User)
var client = new DICOMSCU(local);
var storer = client.GetCStorer(daemon);

var desktopPath = Environment.GetFolderPath(Environment.SpecialFolder.
    Desktop);
var storagePath = Path.Combine(desktopPath, "DICOM␣Storage");

ushort msgId = 1;
var dcmFiles = Directory.GetFiles(storagePath);
foreach (var path in dcmFiles)
```

```
{
    //Reads DICOM object into memory
    var dcm = DICOMObject.Read(path);
    var response = storer.SendCStore(dcm, ref msgId);
    //Write results to console
    Console.WriteLine($"DICOM␣C-Store␣from␣{local.AeTitle}␣=>␣" +
            $"{daemon.AeTitle}␣@{daemon.IpAddress}:{daemon.Port}:" +
            $"{(Status)response.Status}");
}
Console.Read(); //Stop here
```

■

4.4 Conclusion

As you have seen the above examples, the DICOM operations allow for robust automation in the Varian ecosystem. Ultimately, adding these tools to your Varian ecosystem toolkit will can make you a much more powerful physicist/programmer. Previously difficult and time consuming import, export and manipulation operations can become fast and efficient with the capabilities of the DICOM daemons. My hope is that these DICOM lessons have inspired you to automate more and explore new code to create in the fight against cancer.



5. Plotting data with C#

CARLOS ANDERSON, PHD

You may already be familiar with the plotting capabilities of other programming languages, such as MATLAB or R. It may then surprise you to learn that C# does not come with the ability to plot data out of the box. That is not to say that C# and the .NET Framework do not have any graphical capabilities. On the contrary, C# can be used to write sophisticated 2D and 3D graphical applications, including video games. But these capabilities are added to the language by the use of class libraries, which are collections of code that provide some specific functionality.

There are two main class libraries that allow you to plot data in C#: OxyPlot and LiveCharts. Both of these libraries are free and open-source under the MIT license, which means you can use them in commercial applications. They both support many kinds of plots and allow you to customize most aspects of their visualization. OxyPlot has been around since 2010. LiveCharts was started in 2015 by a developer who wanted to have a more "modern" look to his plots. Because OxyPlot is the more established library of the two, I will teach it in this chapter. But the concepts you learn will allow you to easily pick up LiveCharts, if you desire.

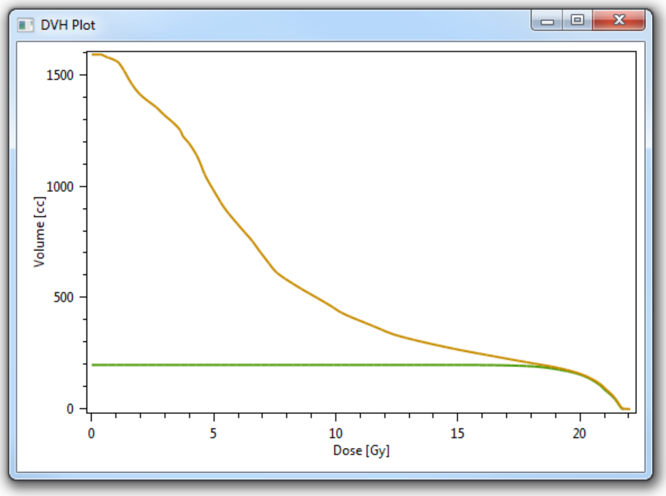
In this chapter, you will learn how to use OxyPlot to plot data and display it to the user as an ESAPI script. You'll start by plotting dose-volume histograms (DVHs) as lines, then you'll improve that script to let the user choose which DVHs to plot. You will learn to change the look of your plot by playing with its axes, colors, and line types. In addition, you'll learn how to export your plot as a PDF file, which you can then add to a report document. Finally, you'll learn how to create column plots, pie plots, and heatmaps, where you'll plot the dose distribution of a patient's plan.

I wrote the scripts in this chapter using ESAPI v. 13.6, but they should work with minimal modifications under ESAPI v. 15. I used Visual Studio Community 2017, which is freely available for download. Also, to easily test scripts within Visual Studio, I used the library EclipsePlugInRunner[1]. This library lets you run your script directly from Visual Studio, without having to open Eclipse every time you want to run it. You don't need EclipsePlugInRunner to follow along, but it'll make testing easier.

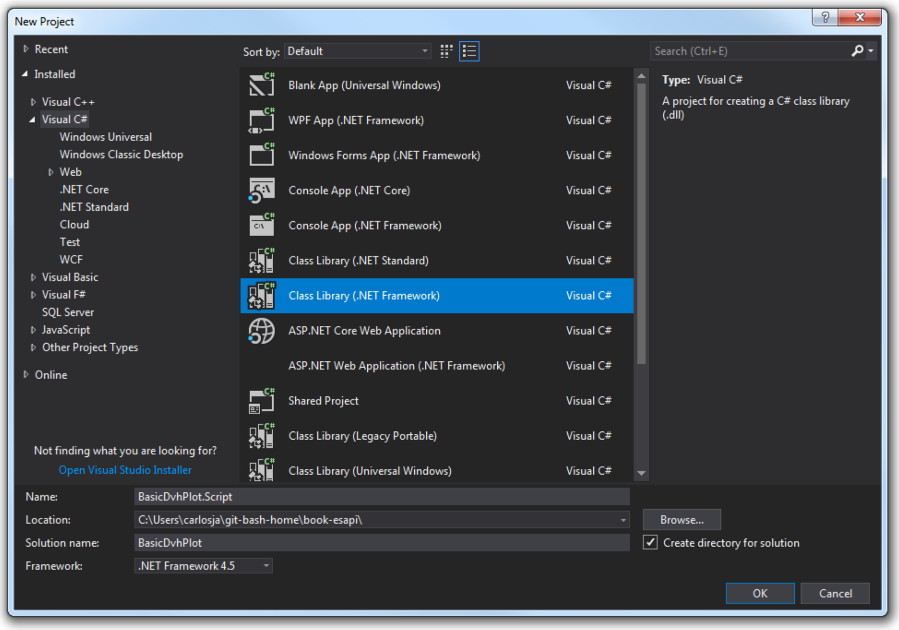
[1]http://www.carlosjanderson.com/run-and-test-plug-in-scripts-from-visual-studio

5.1 Showing a DVH plot

You're going to write an ESAPI plug-in script that will plot the DVHs of specific structures in the opened plan. The plot will automatically allow the user to interact with it, such as displaying the individual data points when the plot is clicked. Here's what your script will look like when it's finished:



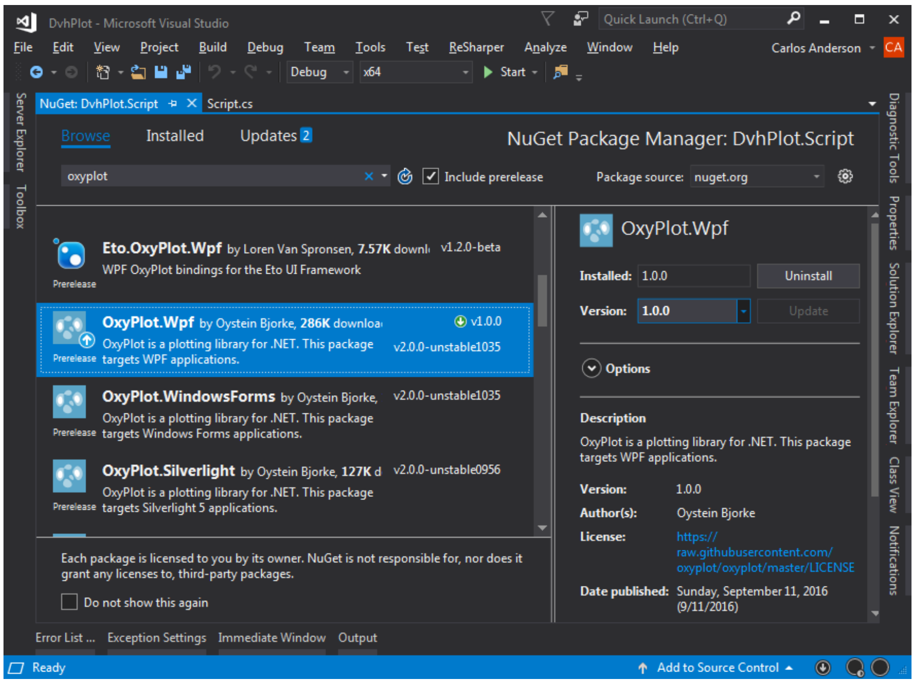
Start by creating a new Class Library (.NET Framework) project in Visual Studio to contain the binary plug-in script (reference ESAPI chapter). Name the project "BasicDvhPlot.Script" and the solution "BasicDvhPlot". The "New Project" dialog box should look like the following (except for the Location, of course):


New Project
Recent
Installed
Visual C++
Visual C#
Windows Universal
Windows Classic Desktop
Web
.NET Core
.NET Standard
Cloud
Test
WCF
Visual Basic
Visual F#
SQL Server
JavaScript
Other Project Types
Online
Not finding what you are looking for?
Open Visual Studio Installer
Sort by: Default
Blank App (Universal Windows) Visual C#
WPF App (.NET Framework) Visual C#
Windows Forms App (.NET Framework) Visual C#
Console App (.NET Core) Visual C#
Console App (.NET Framework) Visual C#
Class Library (.NET Standard) Visual C#
Class Library (.NET Framework) Visual C#
ASP.NET Core Web Application Visual C#
ASP.NET Web Application (.NET Framework) Visual C#
Shared Project Visual C#
Class Library (Legacy Portable) Visual C#
Class Library (Universal Windows) Visual C#
Search (Ctrl+E)
Browse...
OK
Cancel

Type: Visual C#
A project for creating a C# class library (.dll)

Name: BasicDvhPlot.Script
Location: C:\Users\carlosja\git-bash-home\book-esapi\
Solution name: BasicDvhPlot
Framework: .NET Framework 4.5
[✓] Create directory for solution

Now rename the default class to `Script`, and add the references to ESAPI. Also, add references to PresentationFramework, PresentationCore, WindowsBase, and System.Xaml (you'll need them to display a window). Finally, add a reference to the OxyPlot library. To do that, right-click on the References section of your project, and choose Manage NuGet Packages. Your screen should look something like the following:



NuGet Package Manager: DvhPlot.Script

oxyplot

Package source: nuget.org

Eto.OxyPlot.Wpf by Loren Van Spronsen, 7.57K downl v1.2.0-beta
WPF OxyPlot bindings for the Eto UI Framework
Prerelease

OxyPlot.Wpf by Oystein Bjorke, 286K downloa v1.0.0
OxyPlot is a plotting library for .NET. This package targets WPF applications. v2.0.0-unstable1035
Prerelease

OxyPlot.WindowsForms by Oystein Bjorke, v2.0.0-unstable1035
OxyPlot is a plotting library for .NET. This package targets Windows Forms applications.
Prerelease

OxyPlot.Silverlight by Oystein Bjorke, 127K d v2.0.0-unstable0956
OxyPlot is a plotting library for .NET. This package targets Silverlight 5 applications.
Prerelease

Each package is licensed to you by its owner. NuGet is not responsible for, nor does it grant any licenses to, third-party packages.

☐ Do not show this again

OxyPlot.Wpf

Installed: 1.0.0

Version: 1.0.0

Description

OxyPlot is a plotting library for .NET. This package targets WPF applications.

Version: 1.0.0

Author(s): Oystein Bjorke

License: https://raw.githubusercontent.com/oxyplot/oxyplot/master/LICENSE

Date published: Sunday, September 11, 2016 (9/11/2016)

Click on the Browse tab in the main window and type "OxyPlot" in the search box. Find "OxyPlot.Wpf" in the results list, and click on it. On the right window, choose Version 1.0.0 and click on the Install button. You now have the OxyPlot class library reference by your project, which means you can use any of its functionality.

As you know, every ESAPI binary plug-in script must have an `Execute` method in its main class. The `Execute` method is called by Eclipse when your script is started. This method must declare a `ScriptContext` object as a parameter and an optional `Window` object. Add the `Execute` method to the `Script` class with the following code:

Code 5.1

```
public void Excecute(ScriptContext context, Window window)
{
    window.Title = "DVH␣Plot";
    window.Content = CreatePlotView(context.PlanSetup);
}

private PlotView CreatePlotView(PlanSetup plan)
{
    return new PlotView();
}
```

■

R If you're using EclipsePlugInRunner, the code in the `Execute` method should go in the `Run` method. Also, because the `Run` method doesn't have a `context` parameter, pass the `planSetup` variable directly to the `CreatePlotView` method.

Be sure to add `using OxyPlot.Wpf;` at the top of the Script.cs file, where the other `using` statements are. This will make the `PlotView` class available to use in the file. Visual Studio Community 2017 is very good at helping you add the necessary `using` statements as you code. Therefore, I won't continue to remind you to do so when they're needed.

The code above sets the main window's title to "DVH Plot" and assigns the window's content to a new `PlotView` object, which is created in the `CreatePlotView` method. Eclipse will automatically show the window to the user. Because the `PlotView` object is empty, the window will have nothing inside (you'll fix that soon).

The `PlotView` class represents the graphical area (or "view") of the plot. However, you don't directly add the data to be drawn to the `PlotView` object. Instead, you create a `PlotModel` object first, fill it up with data and configure how it should look, and then assign it to the `Model` property of the `PlotView` object. Update the `CreatePlotView` method to create a `PlotModel` object:

Code 5.2
```
private PlotView CreatePlotView(PlanSetup plan)
{
    return new PlotView {Model = CreatePlotModel(plan)};
}

private PlotModel CreatePlotModel(PlanSetup plan)
{
    return new PlotModel();
}
```

Using C#'s "object initialization" feature, the `Model` property can be assigned in the same line that creates the `PlotView` object. At the moment, the new `PlotModel` object is empty. Before you add any DVH curves to this model, however, you need to understand what a *series* is.

A series represents the set of data you want to plot, similar to the concept of a series in an Excel chart. A plot in OxyPlot is composed of one or more series, which may be of different types, such as line, scatter, or column. For this script, you're going to use a line series for each DVH. Modify the `CreatePlotModel` method to create and add each DVH series to the `PlotModel`:

Code 5.3
```
private PlotModel CreatePlotModel(PlanSetup plan)
{
    var model = new PlotModel();
    AddDvhs(model, plan);
    return model;
}

private void AddDvhs(PlotModel model, PlanSetup plan)
{
    var structures = GetDesiredStructures(plan);
    foreach (var structure in structures)
    {
        var dvh = CalculateDvh(plan, structure);
        var series = CreateDvhSeries(dvh);
        model.Series.Add(series);
    }
}
```

The `AddDvhs` method first obtains the desired structures. It then goes through each structure, calculates its DVH, creates a series for it, and finally adds the series to the `PlotModel` object. This method uses several helper methods to accomplish these tasks. It's good programming practice to split up a complex method into smaller, more manageable methods. The first helper method is `GetDesiredStructures`:

Code 5.4
```
private List<Structure> GetDesiredStructures(PlanSetup plan)
{
    var desiredStructureIds = new[] {"PTV", "LIVER"};
    var desiredStructures = new List<Structure>();
    foreach (var structureId in desiredStructureIds)
        desiredStructures.Add(FindStructure(structureId, plan));
    return desiredStructures;
}

private Structure FindStructure(string id, PlanSetup plan)
{
    return plan.StructureSet.Structures.First(s => s.Id == id);
}
```
■

First, the IDs of the structures are hard-coded and stored in an array. (In the next script, the user will be able to choose which DVHs to plot.) Then, an empty list of `Structure` objects is created. For each of the structure IDs in the array, the corresponding `Structure` object is obtained via the `FindStructure` method, and it is added to the list. Finally, the list is returned.

The `FindStructure` method uses LINQ to find the first `Structure` in the plan's structure set whose `Id` property matches the given `id` parameter. LINQ stands for language-integrated query, and it is a C# feature that allows you to easily perform operations on collections, such as lists and arrays. These operations include searching, filtering, sorting, and grouping.

The next helper method, `CalculateDvh`, uses the `GetDVHCumulativeData` method of the `PlanSetup` class to calculate the given structure's DVH in absolute units:

Code 5.5
```
    private DVHData CalculateDvh(PlanSetup plan, Structure structure)
{
    return plan.GetDVHCumulativeData(structure,
        DoseValuePresentation.Absolute,
        VolumePresentation.AbsoluteCm3, 0.01);
}
```
■

This method returns a `DVHData` object, which contains the DVH, and it is passed to the final helper method, `CreateDvhSeries`:

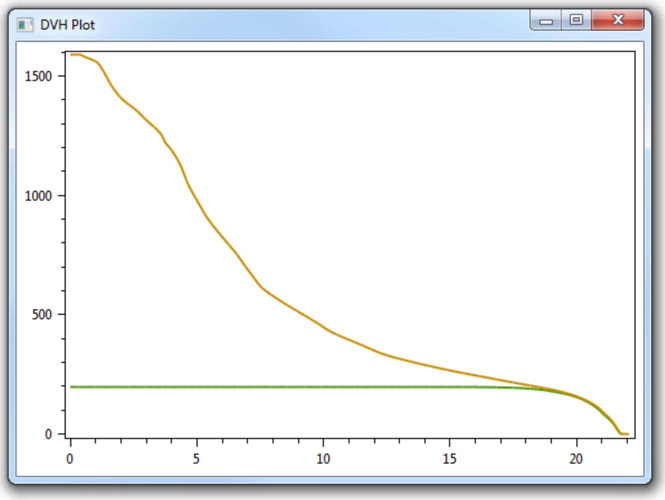
Code 5.6
```
private Series CreateDvhSeries(DVHData dvh)
{
    var series = new LineSeries();
    var points = CreateDataPoints(dvh);
    series.Points.AddRange(points);
    return series;
}

private List<DataPoint> CreateDataPoints(DVHData dvh)
{
    var points = new List<DataPoint>();
    foreach (var dvhPoint in dvh.CurveData)
    {
        var point = CreateDataPoint(dvhPoint);
```

```
            points.Add(point);
        }
        return points;
}

private DataPoint CreateDataPoint(DVHPoint dvhPoint)
{
        return new DataPoint(dvhPoint.DoseValue.Dose, dvhPoint.Volume);
}
```

This method starts by creating the line series for the DVH. It then creates the DVH points using `CreateDataPoint`, described shortly. The created points are then added to the series, and the entire series is returned. Notice that the `Points` property of the series is of type `List<DataPoint>`, so you can use its `AddRange` method to add all the points at once.

In the `CreateDataPoints` method, each point in the DVH curve, provided by `CurveData`, is converted to an OxyPlot `DataPoint` object. The reason you need this conversion is that OxyPlot only understands `DataPoint` objects, not `DVHPoint` objects, which is what `CurveData` contains.

You are now almost ready to run this script in Eclipse (or Visual Studio). First, though, make sure to change the desired structure IDs to something you know your test patient has in its plan. The code you've written so far does not do any error checking or exception handling, so if any structure is not found, it'll crash. Now, you should be able to run the script and see a window like this (of course, the exact DVH curves will be different):



If you're like me, you'd immediately notice the plot is missing something every plot should have: axis titles. Modify the `CreatePlotModel` method to add custom axes:

Code 5.7

```
private PlotModel CreatePlotModel(PlanSetup plan)
{
        var model = new PlotModel();
        AddAxes(model);
        AddDvhs(model, plan);
        return model;
}

private void AddAxes(PlotModel model)
```

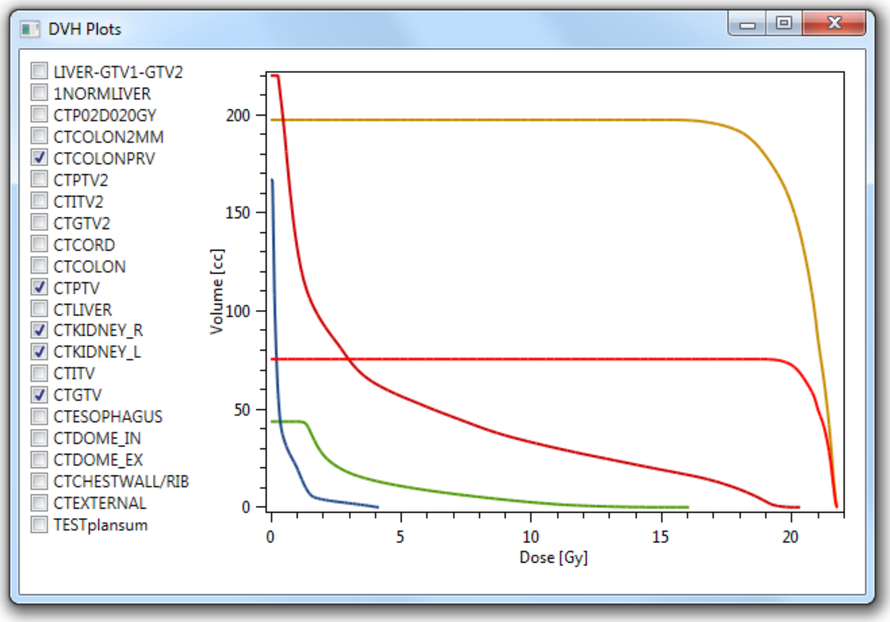
```
{
    // Add x-axis
    model.Axes.Add(new LinearAxis
    {
        Title = "Dose␣[Gy]",
        Position = AxisPosition.Bottom
    });

    // Add y-axis
    model.Axes.Add(new LinearAxis
    {
        Title = "Volume␣[cc]",
        Position = AxisPosition.Left
    });
}
```

The `AddAxes` method first adds the x-axis with the "Dose [Gy]" title, and then the y-axis with the "Volume [cc]" title. The `Position` property specifies where to put each of the axes (you'll learn about other axis positions later in this chapter). If you run the script now, it should look like the screenshot at the start of this section.

5.2 Using XAML and MVVM to display plots

The previous script is pretty useful if you know exactly which structures you want to plot and they match the IDs you hard-coded. But if you ever want to explore different structure DVHs, you'd be out of luck. It would be nice to show the user the structures available in the plan, and let him or her decide which ones to show their DVH.

In this section, you're going to write a new script that does just that. When you're finished, the script will look something like this:



- [ ] LIVER-GTV1-GTV2
- [ ] 1NORMLIVER
- [ ] CTP02D020GY
- [ ] CTCOLON2MM
- [x] CTCOLONPRV
- [ ] CTPTV2
- [ ] CTITV2
- [ ] CTGTV2
- [ ] CTCORD
- [ ] CTCOLON
- [x] CTPTV
- [ ] CTLIVER
- [x] CTKIDNEY_R
- [x] CTKIDNEY_L
- [ ] CTITV
- [x] CTGTV
- [ ] CTESOPHAGUS
- [ ] CTDOME_IN
- [ ] CTDOME_EX
- [ ] CTCHESTWALL/RIB
- [ ] CTEXTERNAL
- [ ] TESTplansum

To accomplish this user interaction, you're going to use Windows Presentation Foundation (WPF). This is a technology by Microsoft that lets you create advanced user interfaces declaratively, using an XML-like language called XAML. I'm not going to go into detail on WPF, but I will show you enough to create the script shown above.

R For more information on WPF (and C# in general), you can read the relevant chapters in the excellent book "Pro C# 7" by Andrew Troelsen. For more practical examples using WPF, see "Windows Presentation Foundation 4.5 Cookbook" by Pavel Yosifovich.

In addition to WPF, you will use the Model-View-ViewModel (MVVM) pattern and data-binding to connect to WPF controls, such as buttons and check boxes. In short, the MVVM pattern separates the view from the data (or model) that the view is supposed to display. So instead of changing the view directly, you change the view model, which is connected to the view via data-binding. Don't worry if you don't fully understand this concept, it'll become clearer once you see some working code.

Just like you did in the previous section, start by creating a new class library project in Visual Studio to contain the binary plug-in script. Name the solution "DvhPlot" and the project "DvhPlot.Script." Rename your main script class to `Script` and use the following `Execute` method:

Code 5.8
```
public void Execute(ScriptContext context, Window window)
{
    var mainViewModel = new MainViewModel(context.PlanSetup);
    var mainView = new MainView(mainViewModel);
    window.Title = "DVH␣Plots";
    window.Content = mainView;
}
```
■

It looks similar to your previous script, except that now you first create the view model, then the view, and finally you assign the view to the window's content. `MainViewModel` and `MainView` don't exist yet, but you'll create these soon. Before you do, however, remember to reference the same libraries as before, including ESAPI and the OxyPlot library.

Now, add a new WPF UserControl to the DvhPlot.Script project by right-clicking on the project, selecting Add and then New Item, and finally choosing "User Control (WPF)" (don't choose "User Control" without the "(WPF)" as it is a different kind of item). Call this new user control "MainView." You should now have two new files in your project: MainView.xaml and MainView.xaml.cs. Open MainView.xaml and modify it to the following:

Code 5.9
```
<UserControl
    x:Class="DvhPlot.Script.MainView"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:oxy="http://oxyplot.org/wpf"
    >

    <Grid Margin="8">
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>

        <ItemsControl
            Grid.Column="0"
```

```
              ItemsSource="{Binding␣Structures}"
              >
              <ItemsControl.ItemTemplate>
                  <DataTemplate>
                      <CheckBox Content="{Binding␣Id}" />
                  </DataTemplate>
              </ItemsControl.ItemTemplate>
          </ItemsControl>

          <oxy:PlotView
              Grid.Column="1"
              Model="{Binding␣PlotModel}"
              />
      </Grid>
  </UserControl>
```

As you can see, this code is not written in C#; it is written in XAML. The syntax is very similar to XML, where elements are placed inside angle brackets. These elements may have properties (like the `Margin` property for the `Grid` element), or may have nested elements (like the `CheckBox` element inside `DataTemplate`). Typically, XAML maps very closely to the physical layout of the user interface. For example, the `PlotView` element is inside the `Grid`, which itself is inside the `UserControl`. As a result, the view will display the `PlotView` laid out in a `Grid` panel within the bounds of the `UserControl`.

The `UserControl` represents a general user-defined control. If there's nothing inside the control, nothing will be displayed. Therefore, one usually has to fill up the user control with other items. Typically, these items are arranged using a panel (or layout) control, such as the `Grid` panel you've defined above. Other panel controls include `StackPanel`, `DockPanel`, and `WrapPanel`.

The `Grid` panel lays out its child controls in a grid or table, arranged in rows and columns. In the code above, you can see that two columns have been defined (the grid automatically has one row). By default, the grid is evenly divided by the number of columns, so that each column has the same width. But by setting the column's `Width` property to `Auto`, as has been done above, the column will only take up as much space as it needs. The remaining columns (in this case, just one) will fill up the rest of the available space.

The first element inside the `Grid` is the `ItemsControl` element. This is a control that shows its child elements in a list. Notice that the `Grid.Column` property has been set to `0`. This means that the `ItemsControl` will appear in the first column (on the left) of the `Grid` panel (remember than indexes in C# start at 0, not 1).

The `ItemsSource` property of the `ItemsControl` defines the items to be displayed as a list. As was mentioned at the start of this section, the user will be presented with a list of structures to choose from. Therefore, the list needs to contain the structures of the opened plan. Here we use the magic of data-binding and simply assign this property to `{Binding Structures}`. The actual data is obtained and stored in the view model, which you'll see shortly.

The structures shouldn't just be presented as is; they need to be displayed as check boxes that the user can interact with. To change the way items inside an `ItemsControl` are displayed, you use a `DataTemplate`. The `DataTemplate` must be defined inside `ItemsControl.ItemTemplate`. The `DataTemplate` above contains a single `CheckBox` element. The check box's content (that is, the text to be displayed) is data-bound to the structure's `Id` property. Visually, you'll see a check box and then the structure's ID for every structure in the list.

The second element in the `Grid` is the `PlotView`. Notice that it's defined with the `oxy:` prefix. This is because `PlotView` comes from an external library, so its namespace needs to be defined above (see the `xmlns:oxy` definition in the `UserControl` element). You can think of it as a `using` statement in XAML. The `PlotView`'s grid column is `1`, so it will appear in the second column of the grid. The `Model` property, which provides the `PlotView` with everything it needs to draw its contents, is data-bound to the `PlotModel` property of the view model.

As you've seen, some of the XAML properties are data-bound to properties of the view model. You may be wondering how the view knows which view model to use. There's no magic here. You need to manually set the view's `DataContext` property to the view model it should use for data-binding. Open the MainView.xaml.cs file and modify it as follows:

Code 5.10

```
public partial class MainView : UserControl
{
    // Create dummy PlotView to force OxyPlot.Wpf to be loaded
    private static readonly PlotView PlotView = new PlotView();

    public MainView(MainViewModel viewModel)
    {
        InitializeComponent();
        DataContext = viewModel;
    }
}
```

Ignore the `PlotView` static field for the moment. In the constructor, you pass in the view model. After the view initializes its components by calling the `InitializeComponent` method, you set the `DataContext` to the view model. (You don't need to write the `InitializeComponent` method because it's already been written for you as part of WPF.) Now, the static field `PlotView` is defined above to force the OxyPlot.Wpf library to be loaded at the right time (it's kind of a bug that this is necessary, so I won't explain this further).

It's time to write the view model, so create a `MainViewModel` class and replace the corresponding source file's contents with the following:

Code 5.11

```
using System.Collections.Generic;
using OxyPlot;
using OxyPlot.Axes;
using VMS.TPS.Common.Model.API;

namespace DvhPlot.Script
{
    public class MainViewModel
    {
        private readonly PlanSetup _plan;

        public MainViewModel(PlanSetup plan)
        {
            _plan = plan;

            Structures = GetPlanStructures();
            PlotModel = CreatePlotModel();
        }

        public IEnumerable<Structure> Structures { get; private set; }

        public PlotModel PlotModel { get; private set; }
        private IEnumerable<Structure> GetPlanStructures()
        {
            return _plan.StructureSet != null
                ? _plan.StructureSet.Structures
                : null;
        }

        private PlotModel CreatePlotModel()
        {
            var plotModel = new PlotModel();
            AddAxes(plotModel);
            return plotModel;
        }

        private static void AddAxes(PlotModel plotModel)
        {
            plotModel.Axes.Add(new LinearAxis
            {
                Title = "Dose␣[Gy]",
                Position = AxisPosition.Bottom
            });

            plotModel.Axes.Add(new LinearAxis
            {
                Title = "Volume␣[cc]",
                Position = AxisPosition.Left
            });
        }
    }
}
```

The view model's constructor receives the `PlanSetup`, which is then stored as a private field. Then, the structures of the plan are extracted and stored in the `Structures` property. This is the same property that is data-bound to the `ItemsControl` in XAML. The `PlotModel` is then created and stored in a property as well. If you remember, this property is data-bound to the `PlotView` in XAML.

The `GetPlanStructures` method simply returns the structures in the opened plan, if available (or `null` if there are none). The `CreatePlotModel` and `AddAxes` methods should look familiar. You wrote these methods in the previous script, where they are used to add the x-axis and y-axis to the plot. Notice that there's no code here that adds the DVH curves, and that's because they are added only after the user chooses the structures of interest (you'll see that code in a bit).

If you compile and run this script in Eclipse (or in Visual Studio if you're using EclipsePlugInRunner), you'll see the list of check boxes for each structure and an empty plot. You're able to select any structure, but the plot is not updated. You still need to write the code to add a DVH curve for each of the structures that the user selects. Open your XAML file (MainView.xaml) and edit the check box to the following:

Code 5.12

```
<CheckBox
    Content="{Binding␣Id}"
    Checked="Structure_OnChecked"
    Unchecked="Structure_OnUnchecked"
    />
```

The `Checked` and `Unchecked` properties are actually *events*. These events are triggered when either the check box is checked or unchecked. The values they're assigned to are the names of methods in the code-behind file (MainView.xaml.cs) that are executed when the associated events occur. Open the MainView.xaml.cs file and change the code to the following:

Code 5.13
```
public partial class MainView : UserControl
{
    // Create dummy PlotView to force OxyPlot.Wpf to be loaded
    private static readonly PlotView PlotView = new PlotView();

    private readonly MainViewModel _vm;

    public MainView(MainViewModel viewModel)
    {
        _vm = viewModel;

        InitializeComponent();
        DataContext = viewModel;
    }

    private void Structure_OnChecked(
        object checkBoxObject, RoutedEventArgs e)
    {
        _vm.AddDvhCurve(GetStructure(checkBoxObject));
    }

    private void Structure_OnUnchecked(
        object checkBoxObject, RoutedEventArgs e)
    {
        _vm.RemoveDvhCurve(GetStructure(checkBoxObject));
    }

    private Structure GetStructure(object checkBoxObject)
    {
        var checkbox = (CheckBox)checkBoxObject;
        var structure = (Structure)checkbox.DataContext;
        return structure;
    }
}
```
■

In the constructor you're now saving the view model object because it'll be used by other methods. In the event handlers (that is, the methods assigned to `Checked` and `Unchecked`), you're calling the add or remove DVH methods in the view model, passing the in the structure the user checked or unchecked. This structure is extracted from the check box itself, using the check box's `DataContext`, which is set to the `Structure` in the list. Unlike the `MainView`'s `DataContext`, you didn't need to set the check box's `DataContext` yourself because the `ItemsControl` automatically sets the `DataContext` of its items (in this case, the check box) to the corresponding item from its `ItemsSource` (in this case, a `Structure` object).

The `AddDvhCurve` and `RemoveDvhCurve` methods don't exist in the view model yet, so open the `MainViewModel` class and add the following code:

Code 5.14
```
public void AddDvhCurve(Structure structure)
{
    var dvh = CalculateDvh(structure);
    PlotModel.Series.Add(CreateDvhSeries(structure.Id, dvh));
    UpdatePlot();
}

public void RemoveDvhCurve(Structure structure)
{
    var series = FindSeries(structure.Id);
    PlotModel.Series.Remove(series);
    UpdatePlot();
}

private DVHData CalculateDvh(Structure structure)
{
    return _plan.GetDVHCumulativeData(structure,
        DoseValuePresentation.Absolute,
        VolumePresentation.AbsoluteCm3, 0.01);
}

private Series CreateDvhSeries(string structureId, DVHData dvh)
{
    var series = new LineSeries {Tag = structureId};
    var points = dvh.CurveData.Select(CreateDataPoint);
    series.Points.AddRange(points);
    return series;
}

private DataPoint CreateDataPoint(DVHPoint p)
{
    return new DataPoint(p.DoseValue.Dose, p.Volume);
}

private Series FindSeries(string structureId)
{
    return PlotModel.Series.FirstOrDefault(x =>
        (string)x.Tag == structureId);
}

private void UpdatePlot()
{
    PlotModel.InvalidatePlot(true);
}
```

The `AddDvhCurve` method first calculates the DVH using a method you've seen before. Then, the DVH series is created and added to the `PlotModel` using a method you've also seen before. There's one important difference, though. The ID of the structure is set as the series's tag. The `Tag` property lets you associate any object with the series. In this case, you associate the structure's ID with a specific series. This will make it easy to find the structure later if the related series needs to be removed. Finally, the `UpdatePlot` method forces the plot to be redrawn by invalidating it. This is the way you cause the `PlotView` to refresh itself, which you need to do after you make any changes to the `PlotModel`.

The `RemoveDvhCurve` method uses the series's tag to get the correct series, and then removes it from the `PlotModel`. To find the series, the `FindSeries` method uses LINQ to return the first item in the `Series` collection whose tag equals the structure's ID. The `Tag` property is cast to a `string` because its type is actually an `object`. Again, you need to call `UpdatePlot` to force the `PlotView` to redraw itself.

The script is now complete. If you run it, you'll see a window similar to that shown at the beginning of this section. When you check on a structure, its DVH appears on the plot. When you uncheck it, it disappears. This is a very simple script, and WPF with MVVM may have added more complexity than if you had worked directly with the UI controls themselves. However, as you increase your script's functionality, separating the view from the data (via view models) will make your code easier to understand and maintain.

5.3 Customizing a plot's look

The default look of the plots you've seen so far are pretty decent. But you may want to improve it still by adding a legend, changing the line colors, axis font size, border thickness, and other plot properties to your liking. In this section, you'll see how OxyPlot lets you make many customizations to change the look of your plots. You're going to modify the script from the last section to play with these customizations.

5.3.1 Legend

By default, a legend is shown on a plot if the titles of the series have been set. You can set each series title when the series is created. Update the `CreateDvhSeries` method to set the title:

Code 5.15
```
private Series CreateDvhSeries(string structureId, DVHData dvh)
{
    var series = new LineSeries
    {
        Title = structureId,
        Tag = structureId
    };
    var points = dvh.CurveData.Select(CreateDataPoint);
    series.Points.AddRange(points);
    return series;
}
```
■

The line series title is set to the ID of the structure it represents. If you run the script now, you'll see the legend displayed on the top right corner of the plot, inside the plot's borders.

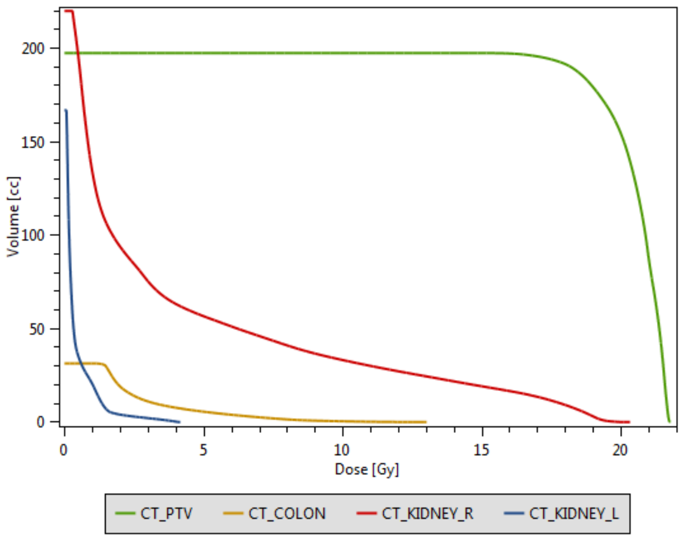
The `PlotModel` object has many properties you can change to customize its look, including the legend. You're going to change the legend's position so that it's shown outside and below the plot horizontally. You'll also add a border around it and change the background to a light gray. Update the `CreatePlotModel` method as follows:

Code 5.16
```
private PlotModel CreatePlotModel()
{
    var plotModel = new PlotModel();
    AddAxes(plotModel);
    SetupLegend(plotModel);
    return plotModel;
}

private void SetupLegend(PlotModel pm)
{
    pm.LegendBorder = OxyColors.Black;
    pm.LegendBackground = OxyColor.FromAColor(32, OxyColors.Black);
    pm.LegendPosition = LegendPosition.BottomCenter;
    pm.LegendOrientation = LegendOrientation.Horizontal;
    pm.LegendPlacement = LegendPlacement.Outside;
}
```

The main change is the addition of the `SetupLegend` method. It sets the legend-related properties of the `PlotModel` to custom values. Some of the types for these properties are enumerations (`enum` in C#), so the possible choices you can set them to are clearly defined (for example, the `LegendPlacement` property can only be `Inside` or `Outside`).

For a property of type `OxyColor`, you can set it to a named color provided by the `OxyColors` class, or to a specific color using any of the various static methods in the `OxyColor` class. One interesting method is `FromAColor`, which takes an *alpha* (or opacity) value and a base color. For example, the `LegendBackground` is set to a base color of black with an alpha value of 32 (or only 12.5% opacity), so it'll show as a very light gray.

If you run the script and select a few structures, the plot should now look something like this:



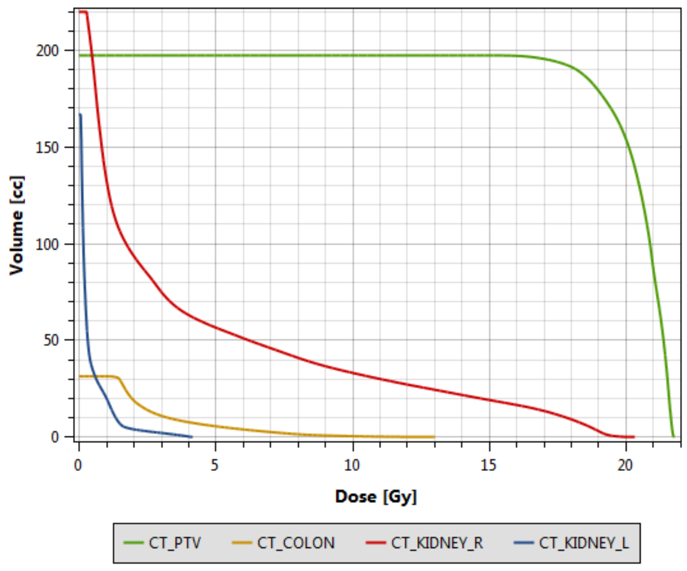
5.3.2 Axes

Currently, the axis titles appear too small and too close to the axes. You're going to increase the font size and make them bold. You'll also add more space between the title and the axes. Finally, you'll add the major and minor grid lines so it's easier to know the dose and volume values for each line. Modify the `AddAxes` method as follows:

Code 5.17

```
private static void AddAxes(PlotModel plotModel)
{
    plotModel.Axes.Add(new LinearAxis
    {
        Title = "Dose␣[Gy]",
        TitleFontSize = 14,
        TitleFontWeight = FontWeights.Bold,
        AxisTitleDistance = 15,
        MajorGridlineStyle = LineStyle.Solid,
        MinorGridlineStyle = LineStyle.Solid,
        Position = AxisPosition.Bottom
    });
    plotModel.Axes.Add(new LinearAxis
    {
        Title = "Volume␣[cc]",
        TitleFontSize = 14,
        TitleFontWeight = FontWeights.Bold,
        AxisTitleDistance = 15,
        MajorGridlineStyle = LineStyle.Solid,
        MinorGridlineStyle = LineStyle.Solid,
        Position = AxisPosition.Left
    });
}
```

To get your plot to look right, you may need to play around with different values. Unfortunately, the OxyPlot documentation does not always tell you what the default values are. For example, it may not be clear that a font size of 14 is an increase in size until you try it. Here's what the plot looks like after making these changes:



There are many more axis settings you can change, such as the major and minor step sizes, the minimum and maximum axis limits, whether to reverse the axis, the positions of the tick marks (inside, center, or outside), and the colors for the title, axes, tick marks, and grid lines. I suggest you explore the available properties in Visual Studio.

5.3.3 Plot area

To show you that OxyPlot is highly configurable, you're going to make the plot look more like Eclipse's DVH plot. In other words, you're going to change the plot's background color to black and match each DVH line color to the corresponding structure's color in Eclipse. You're also going to change the line thickness and type to emphasize certain structures.

The `PlotModel` class has the properties `Background` and `PlotAreaBackground`. `Background` refers to the entire figure, including the axes title, labels, and plot. `PlotAreaBackground` refers to only the plot area. Modify the `CreatePlotModel` method to make the plot area 90% black (I found that 100% black was too strong):

Code 5.18
```
private PlotModel CreatePlotModel()
{
    var plotModel = new PlotModel
    {
        PlotAreaBackground = OxyColor.FromAColor(230, OxyColors.Black)
    };
    AddAxes(plotModel);
    SetupLegend(plotModel);
    return plotModel;
}
```
■

Because the background is so dark now, you need to lighten the color of the grid lines; otherwise, they won't be visible. Update the `AddAxes` method as follows:

Code 5.19
```
private static void AddAxes(PlotModel plotModel)
{
    plotModel.Axes.Add(new LinearAxis
    {
        Title = "Dose␣[Gy]",
        TitleFontSize = 14,
        TitleFontWeight = FontWeights.Bold,
        AxisTitleDistance = 15,
        MajorGridlineColor = OxyColor.FromAColor(64, OxyColors.White),
        MinorGridlineColor = OxyColor.FromAColor(32, OxyColors.White),
        MajorGridlineStyle = LineStyle.Solid,
        MinorGridlineStyle = LineStyle.Solid,
        Position = AxisPosition.Bottom
    });

    plotModel.Axes.Add(new LinearAxis
    {
        Title = "Volume␣[cc]",
        TitleFontSize = 14,
        TitleFontWeight = FontWeights.Bold,
        AxisTitleDistance = 15,
        MajorGridlineColor = OxyColor.FromAColor(64, OxyColors.White),
        MinorGridlineColor = OxyColor.FromAColor(32, OxyColors.White),
        MajorGridlineStyle = LineStyle.Solid,
        MinorGridlineStyle = LineStyle.Solid,
        Position = AxisPosition.Left
    });
}
```
■

The main change is to add the grid line colors. They're now based on white with an opacity of about 25% for the major grid lines and 12.5% for the minor grid lines. The default was based on black with the same opacities.

Now you're going to change the color, thickness, and style of the DVH lines. The color will be obtained from ESAPI since every `Structure` object has a `Color` property. The thickness and style will be determined from the contents of the structure ID. The rule is that if the structure ID contains "PTV," it should have a thicker line. If it contains "_R" (representing the organ on the right side), it should have a dashed line. This will help differentiate it from the left organ since they are often the same color in Eclipse. Update the `CreateDvhSeries` method to the following:

Code 5.20

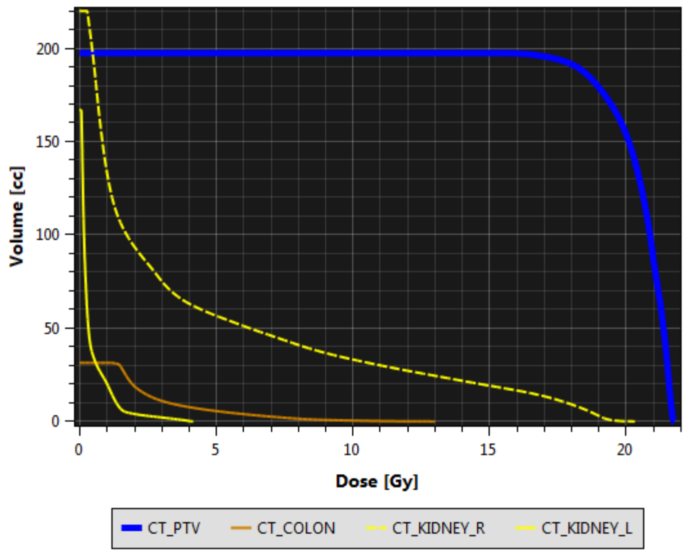
```
private Series CreateDvhSeries(string structureId, DVHData dvh)
{
    var series = new LineSeries
    {
        Title = structureId,
        Tag = structureId,
        Color = GetStructureColor(structureId),
        StrokeThickness = GetLineThickness(structureId),
        LineStyle = GetLineStyle(structureId)
    };
    var points = dvh.CurveData.Select(CreateDataPoint);
    series.Points.AddRange(points);
    return series;
}

private OxyColor GetStructureColor(string structureId)
{
    var structures = _plan.StructureSet.Structures;
    var structure = structures.First(x => x.Id == structureId);
    var color = structure.Color;
    return OxyColor.FromRgb(color.R, color.G, color.B);
}

private double GetLineThickness(string structureId)
{
    if (structureId.ToUpper().Contains("PTV"))
        return 5;
    return 2;
}

private LineStyle GetLineStyle(string structureId)
{
    if (structureId.ToUpper().Contains("_R"))
        return LineStyle.Dash;
    return LineStyle.Solid;
}
```

The `Color` property of the `LineSeries` is assigned to the color returned by the `GetStructureColor` method. In this method, the ESAPI structure is obtained from the plan, as you've seen before. Event though the structure has a `Color` property, you can't return it directly because it's not of the expected `OxyColor` type. To convert it to an `OxyColor`, you use the `FromRgb` method to create a new `OxyColor` object with its red, green, and blue components taken from the structure's color.

Next, the `StrokeThickness` of the line is determined using the `GetLineThickness` method. This method checks whether the structure ID contains the string "PTV." Notice that the ID is first converted to uppercase in order to account for case differences (you want structure IDs that contain "Ptv," "ptv," etc. to be considered). The underlying structure ID is not actually changed, as the `ToUpper` method only returns the uppercase string but doesn't change it. If the structure ID does contain "PTV", a thickness of 5 is returned. For all other structures, a thickness of 2 is returned.

Finally, the `LineStyle` property is assigned using the `GetLineStyle` method. It returns a line style of `Dash` if the structure ID (uppercase) contains "_R". Otherwise, it returns the a `Solid` line style. As mentioned, the "_R" is one convention for specifying the structure on the right side. Your clinic may use a different convention, such as "R_" or "Right-".

If you run the script now, the plot should look like this:



OxyPlot has many properties you can change to customize your plot the way you want. You've only seen a fraction of these properties. I encourage you to explore the `PlotModel`, `Axis`, and `LineSeries` classes to see what else you can customize. The other types of series (like column and heatmap, which you'll see later in this chapter) also contain their own set of properties.

5.4 Exporting a plot for reporting

It's often the case that you want to export your plot to include it as part of a report document. You can do this using OxyPlot's PDF exporting functionality. In this section, you're going to expand on the DVH plotting script to let the user right-click on the plot, choose to export it as a PDF, and generate a PDF containing the plot. Open the MainView.xaml file and change the `PlotView` element to the following:

Code 5.21
```
<oxy:PlotView
    Grid.Column="1"
    Model="{Binding␣PlotModel}"
    >
    <oxy:PlotView.ContextMenu>
        <ContextMenu>
            <MenuItem
                Header="Export␣to␣PDF..."
                Click="ExportPlotAsPdf"
            />
        </ContextMenu>
    </oxy:PlotView.ContextMenu>
</oxy:PlotView>
```

The main change is the addition of a `ContextMenu` element inside the `PlotView`. The `ContextMenu` defines a list of `MenuItems` that will appear when the user right-clicks on the plot. The only item is the "Export to PDF...", which calls the `ExportPlotAsPdf` method when the item is clicked.

The `ExportPlotAsPdf` needs to be defined in the MainView.xaml.cs file. Open it and add the method as follows:

Code 5.22
```
private void ExportPlotAsPdf(object sender, RoutedEventArgs e)
{
    var filePath = GetPdfSavePath();
    if (filePath != null)
        _vm.ExportPlotAsPdf(filePath);
}

private string GetPdfSavePath()
{
    var saveFileDialog = new SaveFileDialog
    {
        Title = "Export␣to␣PDF",
        Filter = "PDF␣Files␣(*.pdf)|*.pdf"
    };

    var dialogResult = saveFileDialog.ShowDialog();

    if (dialogResult == true)
        return saveFileDialog.FileName;
    else
        return null;
}
```

First, the output file path (that is, location) is obtained from the user using the `GetPdfSavePath` helper method (more on this below). If the file path is not `null`, the `ExportPlotAsPdf` method of the view model is called, passing the path where the PDF file should be saved to.

In the `GetPdfSavePath` method, the user is shown the typical save dialog box. You don't need to create this dialog box yourself because it already exists in the `Microsoft.Win32` namespace. To use it, you need to create a new instance of `SaveFileDialog` and specify any relevant properties. Here, you set the dialog box's title and the path filter. The filter causes only PDF files to be shown in the dialog, and when you type your path without the ".pdf" extension, it is automatically added.

Next, the dialog box is shown using the `ShowDialog` method. After the user chooses where to save the file and clicks OK, the `ShowDialog` method returns `true`. If this is the case, the `FileName` property of the dialog box, which stores the file path that the user selected, is returned. If the user clicks on the Cancel button on the dialog box, the method returns `null`. (This is why the file path was checked whether it was `null` in the `ExportPlotAsPdf` method.)

The `ExportPlotAsPdf` method in the `MainView` hasn't actually done much, other than get the file path from the user. The real work is done by the `ExportPlotAsPdf` method in the `MainViewModel`. Open the MainViewModel.cs file and add the following method:

Code 5.23
```
public void ExportPlotAsPdf(string filePath)
{
    using (var stream = File.Create(filePath))
    {
        PdfExporter.Export(PlotModel, stream, 600, 400);
    }
}
```

First, a `Stream` object is created from the file path and wrapped in a `using` statement. A *stream* provides a generic way of accessing a sequence of bytes, such as a file. The `using` statement manages when to close and dispose of the stream when you're done using it. You can learn more about these concepts in any c# reference book.

The `PdfExporter` class from OxyPlot has a static `Export` method. You pass it the plot model, the stream, and the desired width and height of the plot. It then creates a PDF version of the plot and saves it to the stream (which in this case is a file). Notice that you didn't need to pass the `PlotView` to the `Export` method, only the `PlotModel`. This is because the `PlotModel` contains everything that should be in the plot. `PlotView` uses it to render the plot on the screen; `PdfExporter` uses it to export it to a file.

Now, run the script. Choose some DVHs to plot, and then right-click on the plot. Choose to export as a PDF, and then specify the location to save it to. When done, you'll be able to open the PDF file containing the plot. You can then use this PDF to add to a report document.

R OxyPlot also comes with the classes `PngExporter` and `SvgExporter` to let you export your plot as a PNG or SVG file.

5.5 Working with various plot types

So far, you've used the `LineSeries` object to plot lines for each structure's DVH. OxyPlot also supports other kinds of plots, including scatter, column, pie, area, heat map, and contour. Refer to the OxyPlot documentation for the full list of supported plot types. I also recommend downloading the OxyPlot source code and running the sample application. It showcases many of the plot types and features of OxyPlot.

In this section, you're going to create three different plots: column, pie, and heat map. Because you're already familiar with the basic code that gets the plot shown in a window, you won't see it again here. Instead, you'll only see the relevant code for plotting a specific kind of plot. You can easily modify the scripts from previous sections to show these new plots.

5.5.1 Column

The column plot shows you the data as vertical columns. In this example, you're going to plot the meterset units (MU) for each of the beams in a plan. This data is readily available from ESAPI.

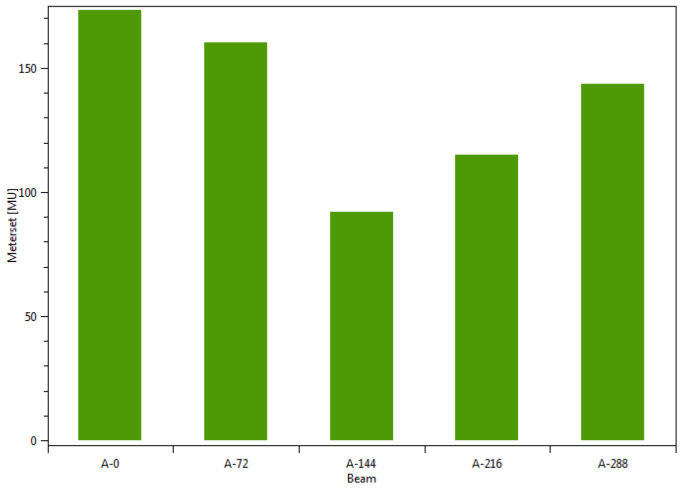
Code 5.24
```
private PlotModel CreatePlotModel()
{
    var plotModel = new PlotModel();
    AddAxes(plotModel);
    AddSeries(plotModel);
    return plotModel;
}

private void AddAxes(PlotModel plotModel)
{
    var xAxis = new CategoryAxis
        {Title = "Beam", Position = AxisPosition.Bottom};
    var beams = _plan.Beams.Where(b => !b.IsSetupField);
    var beamIds = beams.Select(b => b.Id);
    xAxis.Labels.AddRange(beamIds);
    plotModel.Axes.Add(xAxis);

    var yAxis = new LinearAxis
        {Title = "Meterset␣[MU]", Position = AxisPosition.Left};
    plotModel.Axes.Add(yAxis);
}
```

```
private void AddSeries(PlotModel plotModel)
{
    var series = new ColumnSeries();
    var beams = _plan.Beams.Where(b => !b.IsSetupField);
    var items = beams.Select(b => new ColumnItem(b.Meterset.Value));
    series.Items.AddRange(items);
    plotModel.Series.Add(series);
}
```

The column plot requires a category axis, which is specified by adding a `CategoryAxis` object to the plot's axes. The label of each category should be the ID of the beam. To do that, you first obtain the beams that are not "setup" beams, and then extract their IDs. These IDs are then added to the `Labels` property of the `CategoryAxis`.

To create the series, you first create a `ColumnSeries` object. Again, you are only interested in the non-setup beams from the plan. For each beam, you create a `ColumnItem`, which you initialize with the beam's MU value. Finally, these items are added to the `Items` property of the series.

The resulting plot looks like this:



This is the default look. Remember that you can change almost anything about a plot. For column plots, you can change things like their color, outline, and the space between columns. There is also a `BarSeries` object that shows the columns horizontally rather then vertically.

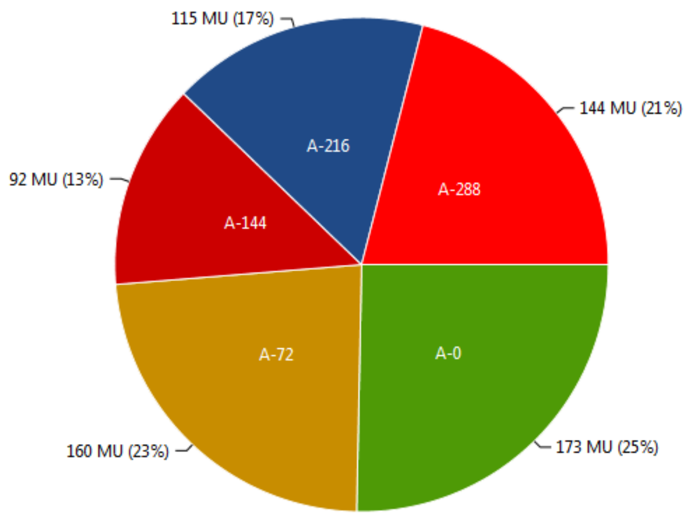
5.5.2 Pie

If you're interested in the contribution of each beam to the total MU, you can show the data above as a pie. The pie plot doesn't need any axes, so remove the call to `AddAxes` in the `CreatePlotModel` method. Then, update the `AddSeries` method to the following:

Code 5.25
```
private void AddSeries(PlotModel plotModel)
{
    var series = new PieSeries
    {
        InsideLabelColor = OxyColors.White,
        OutsideLabelFormat = "{0:f0}␣MU␣({2:f0}%)"
    };
    var beams = _plan.Beams.Where(b => !b.IsSetupField);
```

```
        var slices = beams.Select(b => new PieSlice(b.Id, b.Meterset.Value))
            ;
        foreach (var slice in slices)
            series.Slices.Add(slice);
        plotModel.Series.Add(series);
}
```

First, you create a `PieSeries` object and specify that the labels should be white (the default black labels don't contrast well against the colors of the pie slices). I'll discuss the `OutsideLabelFormat` property later. As before, you obtain the non-setup beams. Next, you create a `PieSlice` object for each beam, passing it a label (in this case, the beam ID) and a value (in this case, the MU). You then add each slice to the series and finally add the series to the `PlotModel`. Here's what the plot looks like now:



The format of the outside labels is specified by the `OutsideLabelFormat` property of the `PieSeries` object. The format is specified as a *composite format string*, which uses placeholders that are replaced when displayed to the user. The `OutsideLabelFormat` recognizes three placeholders: `{0}`, which represents the numerical value, `{1}`, which represents the textual label, and `{2}`, which represents the percentage. For beam A-0, for example, `{0}` would be replaced with 173, `{1}` would be replaced with "A-0," and `{2}` would be replaced with 25%. When you don't specify an `OutsideLabelFormat`, the label only shows the percentage.

The placeholders representing numerical values can themselves be formated to show or hide their decimal parts. For example, if you want to show the MU with one decimal number, you'd use `"{0:f1}"`. For no decimal numbers, you'd use `"{0:f0}"`. Therefore, the format used in the code, `"{0:f0}␣MU␣({2:f0}%)"`, says to include the MU value, the unit (MU), and in parenthesis the percentage, ending in a % symbol.

5.5.3 Heat map

The heat map shows you the data in a matrix as a color wash, similar to Eclipse's dose color wash. In fact, the following example will show you how to plot the dose distribution of a plan as a color wash. In addition, you'll be able to choose the slice (or plane) you want to see using a slider control.

Starting with the DvhPlot script you've worked on previously, modify the MainView XAML to add a slider:

Code 5.26
```
<DockPanel
    Grid.Column="1"
    >
    <Slider
        DockPanel.Dock="Top"
        Value="{Binding␣PlaneIndex}"
        Minimum="0"
        Maximum="{Binding␣MaximumPlaneIndex}"
        Orientation="Horizontal"
        />
    <oxy:PlotView
        DockPanel.Dock="Top"
        Model="{Binding␣PlotModel}"
        />
</DockPanel>
```

The `DockPanel` wraps the `Slider` and the `PlotView`. The `DockPanel` is another kind of panel, like the `Grid`, but allows its items to be docked or stacked in a specific way. An interesting property of the `DockPanel` is that, by default, the last element is resized to fill any remaining space. Because the `PlotView` is the last element, it will be resized automatically as the window is resized.

The `Slider` will let the user choose the plane index of the dose matrix. The `Value` of the slider, is data-bound to the `PlaneIndex` property of the view model. This means that as the user moves the slider, the `PlaneIndex` will be updated automatically. The minimum and maximum values of the slider will represent the lowest and highest plane indexes. Because the maximum plane index isn't known until the script is running, the `Maximum` property is bound to `MaximumPlaneIndex`, which is determined from the dose matrix at runtime.

The heat map requires a `LinearColorAxis`, which describes how the colors will be shown. Open the `MainViewModel` class, and update the `AddAxes` method as follows:

Code 5.27
```
private void AddAxes(PlotModel plotModel)
{
    var xAxis = new LinearAxis
        {Title = "X", Position = AxisPosition.Bottom,};
    plotModel.Axes.Add(xAxis);

    var yAxis = new LinearAxis
        {Title = "Y", Position = AxisPosition.Left};
    plotModel.Axes.Add(yAxis);

    var zAxis = new LinearColorAxis
    {
        Title = "Dose␣[Gy]",
        Position = AxisPosition.Top,
        Palette = OxyPalettes.Rainbow(256),
        Maximum = 33.1
    };
    plotModel.Axes.Add(zAxis);
}
```

The x- and y-axes are the same as before, except for the titles. The z-axis, which represents the range of dose values as a color scale, will be shown above the plot (`AxisPosition.Top`). The color scale can use any number of colors (or *palette*) to represent the range of values. In this case, the

Palette property is set to Rainbow, where violet and blue represent low values, and red represent high values (with all other colors in between). The number passed to Rainbow is the number of color levels to use. Finally, the maximum value of the axis scale is specified. Normally, this value is determined automatically from the data, But because the data will change as the slider is moved, the maximum value needs to be fixed for the entire dose matrix.

Now, modify the AddSeries method to add the heat map:

Code 5.28

```
private void AddSeries(PlotModel plotModel)
{
    plotModel.Series.Add(CreateHeatMap());
}

private Series CreateHeatMap()
{
    return new HeatMapSeries
    {
        X0 = 0, X1 = _plan.Dose.XSize - 1,
        Y0 = 0, Y1 = _plan.Dose.YSize - 1,
        Data = GetDoseData()
    };
}

private double[,] GetDoseData()
{
    _plan.DoseValuePresentation = DoseValuePresentation.Absolute;
    var dose = _plan.Dose;
    var data = new int[dose.XSize, dose.YSize];
    dose.GetVoxels((int)PlaneIndex, data);
    return ConvertToDoseMatrix(data);
}

private double[,] ConvertToDoseMatrix(int[,] ints)
{
    var dose = _plan.Dose;
    var doseMatrix = new double[dose.XSize, dose.YSize];
    for (int i = 0; i < dose.XSize; i++)
        for (int j = 0; j < dose.YSize; j++)
            doseMatrix[i, j] = dose.VoxelToDoseValue(ints[i, j]).Dose;
    return doseMatrix;
}
```

■

The CreateHeatMap method instantiates a new HeatMapSeries object. The x and y limits of the axes need to be specified. These correspond to the dose matrix's limits, which can be obtained using ESAPI. For simplicity, the upper limits are set to the last voxel in the corresponding axis, but you can modify this to physical units (such as cm).

The Data property expects a two-dimensional array of type double, which is created from the dose matrix in the GetDoseData method. First, the DoseValuePresentation is set to Absolute to ensure the doses are in absolute units. Then, the GetVoxels method of the Dose object is called to obtain the dose matrix at the current plane index. The current plane index is stored in the PlaneIndex property (more on that later). Finally, the raw voxel values are converted to dose values using the ESAPI method VoxelToDoseValue.

The final piece of code is to add define the properties the slider is bound to:

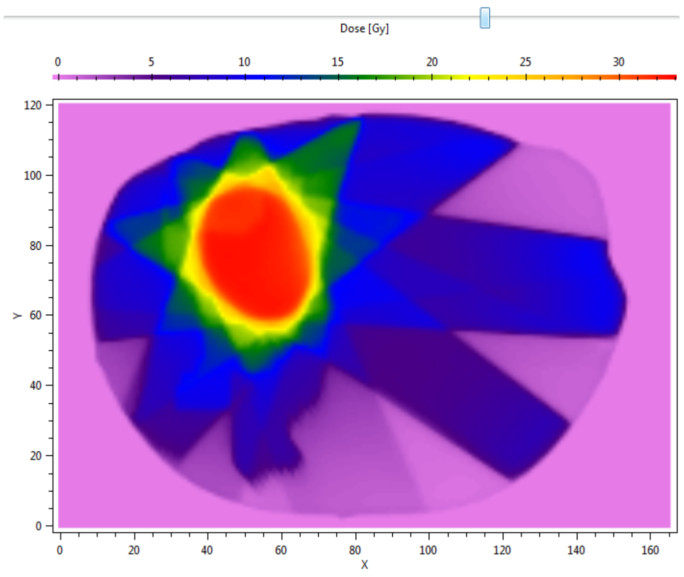
Code 5.29

```
private double _planeIndex;
public double PlaneIndex
{
    get { return _planeIndex; }
    set
    {
        _planeIndex = value;
        PlotModel.Series[0] = CreateHeatMap();
        PlotModel.InvalidatePlot(false);
    }
}

public int MaximumPlaneIndex
{
    get { return _plan.Dose.ZSize - 1; }
}
```

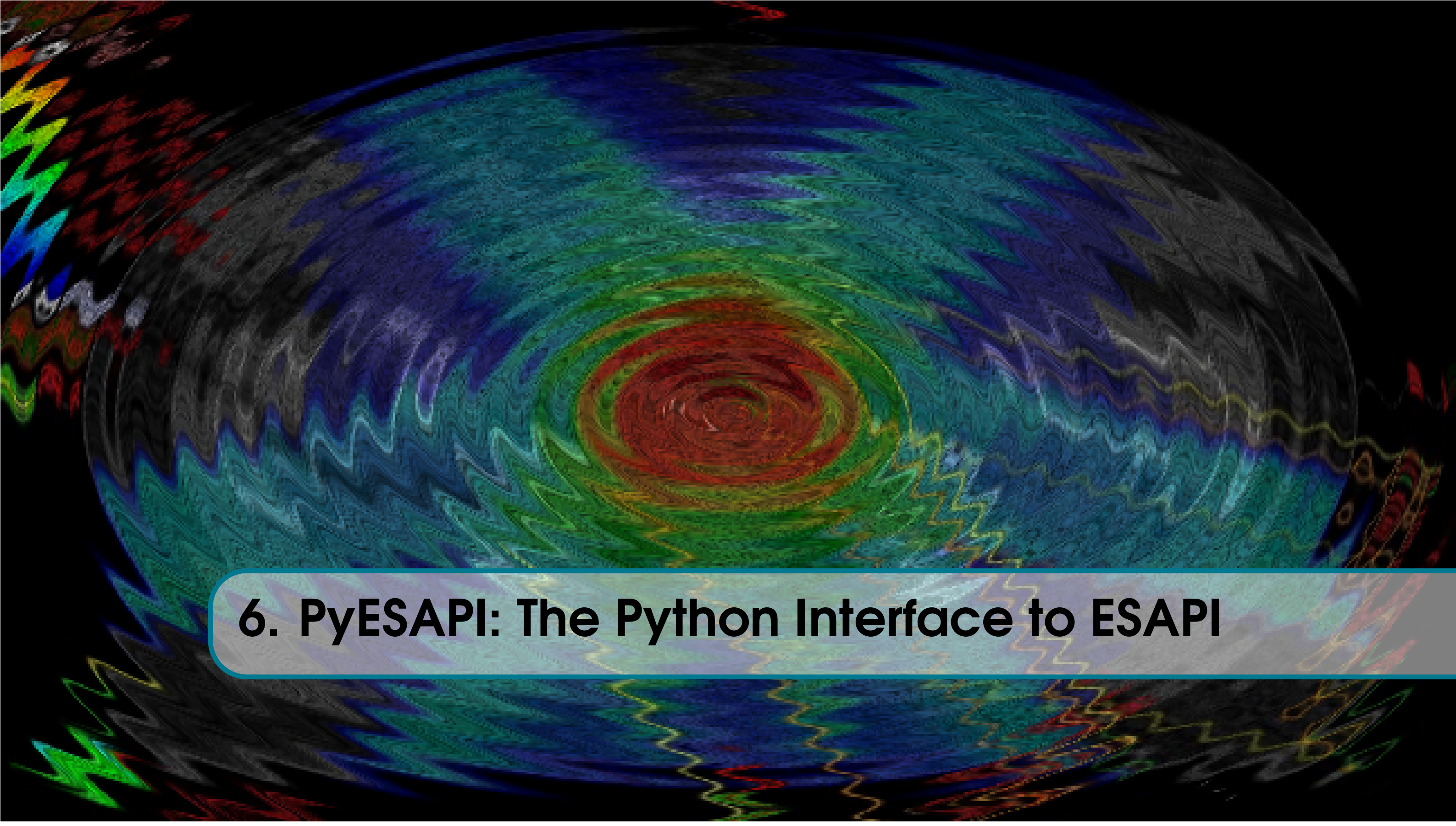
The `PlaneIndex` property represents the index of the current plane the plot is showing. When it's changed using the slider, the property is changed via its `set` method. When this happens, the entire heat map is re-created because a new dose plane needs to be calculated. The plot model is then invalidated in order to redraw the plot.

The `MaximumPlaneIndex` simply returns the last plane index of the dose. This lets the slider set the correct range of allowed values the user can change to. It is data-bound to the `Maximum` property of the slider.

If you run the script, the final plot should look like the following:



Notice the slider control above the plot. As you drag it, the heat map is updated with the correct dose plane.



6. PyESAPI: The Python Interface to ESAPI

Michael M. Folkerts, PhD Candidate

6.1 Introduction

Under the hood, the Eclipse Scripting API (ESAPI) is a curated .NET Framework interface to Eclipse internals. This means an instance of the .NET Framework Common Language Runtime (CLR) is running when you use ESAPI. Starting with Eclipse version 15.5, permission was granted for external binaries to host ESAPI through the .NET CLR, allowing for read-only access in clinical mode, and read/write access in research mode. A project known as Python for .NET implements such a CLR host in Python (actually in CPython). This project allows "direct" access to the .NET objects in ESAPI through native Python.

Interfacing with ESAPI using "real" Python is a real boon for researchers! It puts the power of Eclipse side-by-side with the wealth of Python libraries available, enabling rapid prototyping of pre-clinical research algorithms using mature and popular tools like NumPy, SciPy, Matplotlib, SciKitLearn, Pandas, and TensorFlow.

One of the most exciting Python tools is the Jupyter Notebook. Jupyter enables real-time interactive access to the Python runtime in a user-friendly notebook-style web interface. Using the Python interface to ESAPI in a Jupyter notebook enables interactive access to Eclipse. This allows one to browse the ESAPI data model, draft and debug code, and tune algorithms in real time! The need to re-compile and re-start Eclipse to update/re-run your code is eliminated.

The marriage of Python to C#.NET is not seamless. For example, Python for .NET only automatically converts common .NET types to Python objects on demand (Int, Double, String, etc.). Some work still needs to be done to convert more complex types, like 3D image data, to usable Python objects, such as the ubiquitous NumPy ndarray. This is where PyESAPI (pronounced "pie-sappy") comes in: PyESAPI is a Python library containing a set of tools and shortcuts to bridge the gap between native C#.NET objects and the common Python objects researchers are familiar with. For example, PyESAPI contains methods to extract 3D CT images, dose distributions, and structure segment volumes at multiple resolutions into NumPy ndarrays. PyESAPI also comes with some bonus features like a fast dose approximation algorithm, and an Overlapping Volume

Histogram (OVH) generator. Given in the following sections are examples of how to use PyESAPI for basic research oriented tasks.

6.2 Getting Started

It is recommended to install PyESAPI (and dependencies) on a TBOX with the database set to research mode and use Anaconda Python as your python runtime. Ananconda comes with many of the popular Python libraries mentioned above, pre-built and multi-processor accelerated.

6.2.1 Installation

Start by downloading and installing Anaconda3 (64-bit) for "Just Me". This gives you a default root Anaconda environment in your user's home folder, allowing you to install whatever you need without administrator privileges. Once Anaconda is installed you can install PyESAPI by executing the following command in the "Anaconda Prompt" (search in Windows Start menu):

```
> pip install git+https://github.com/varianapis/pyesapi
```

If your institution uses a proxy for internet access, you might want to add: `--proxy https://username:password@proxy.some.edu:3128` with your specific info to the above command.

6.2.2 Jupyter Notebook

You can start Jupyter Notebook from the Anaconda Prompt by executing:

```
> jupyter notebook
```

Alternatively, you can launch it from the Windows start menu by searching for "Jupyter Notebook". The remainder of this chapter continues in the Jupyter Notebook environment. One can still create normal Python scripts containing the same code we show below, however, you might want to save the plots rather than show them.

One big benefit of Jupyter Notebook (and the IPython kernel) is the tab-completion/intellesense-like functionality. When working in a notebook, once you create an object, you can then type "." and `Tab` to see a list of members. You can also type `Shift+Tab` while inside a function call to see its parameters and docstring.

6.2.3 Import PyESAPI and Start the ESAPI Application

The following code cell will import PyESAPI as `pyesapi` and start an ESAPI Application labeled "python-demo" (used for logging purposes). It is important to include the `atexit` functionality to cleanly shutdown the ESAPI Apllication when the Jupyter Notebook kernel is killed or restarted. Failure to do so will result in a (gentle) crash.

Code 6.1
```
import pyesapi
import atexit
# load app only once
app = pyesapi.CustomScriptExecutable.CreateApplication('python_demo')
# setup clean exit
atexit.register(app.Dispose);
```
■

6.2.4 Navigating the ESAPI Data Model

Find a Patient

Code 6.2 — Printing list of patients.

```
for pat_sum in app.PatientSummaries:
    print(pat_sum.Id, pat_sum.LastName, pat_sum.FirstName)

## output ##
#  007 Doe John
#  CSI_01 Demo CSI_01
#  EC-034 Brain mets Multiple
#  Eclipse 07 HyperArc Brain
#  Eclipse 11 MCO Brain
#  Eclipse 06 MCO Head_Neck
#  zz_cerv_nod_5 zz_cerv_nod_5
#  RapidPlan-01 RapidPlan Prostate T1cN0M0 PSA21.6 G7
#  RapidPlan-03 RapidPlan HN 3 PTV
#  PH PHA QUA
#  Civco_couch Civco Couch
#  Encompass Encompass SRS
#  Sinmed Sinmed Couch
#  VSS 06 Brain Mets 4 mets, CT-MR
#  VSS 07 Brain Mets 12 Met, CT-MR
#  Eclipse-03 Head and Neck Two PTV's
#  Eclipse-04 Lung Right Upper
#  RapidPlan-02 RapidPlan ProstateNodes T2cN0M0
#  RapidPlan-04 RapidPlan HN 3 PTV
#  RapidPlan-06 RapidPlan Lung LUL
#  Registration 2 Registration Brain CT-MR
#  Registration 5 PET CT to Plan CT Deformable
#  Registration 6 Prostate Target Registration Error
#  SmartSeg 1 Prostate T2 N0 M0 PSA8 GS8
#  SmartSeg 2 Tonsil T3 N2b M0 right
#  SmartSeg 3 Nasopharynx T2 N2 M0 right
#  SmartSeg 4 Rectum T3 N1 M0
#  Eclipse-01 4D Lung PET CT
#  Registration 1 Head and Neck CT and CBCT
#  Registration 3 HeadNeck CT MRI
#  VSS 01 Spine T6 and T11, CT-MR
#  VSS 02 Lung RT and LT Lesions, CT
#  VSS 04 Brain, Primary Meningioma, CT-MR
```

You can then choose an Id from the list and open that patient.

Code 6.3

```
patient = app.OpenPatientById('RapidPlan-01')
print(('Name: 'Name: {patient.FirstName}, LastName: {patient.LastName}'.
    format(**locals()))
# which is equivalent to new python 3 f-string sugar
print(f'Name: {patient.FirstName}, LastName: {patient.LastName}')

## output ##
#  Name: Prostate T1cN0M0 PSA21.6 G7, LastName: RapidPlan
#  Name: Prostate T1cN0M0 PSA21.6 G7, LastName: RapidPlan
```

A little about Lots

Python for .NET, or simply `pythonnet`, wraps collection types in a primitive iterator.

Code 6.4
```
# python list comprehension on pythonnet collection iterator
[c.Id for c in patient.Courses]

## output ##
#  ['C1']
```

The Lot object is a custom collection container built into PyESAPI to make up for the lack of support for extension methods in Python for .NET. The Lot object was designed to function similar to C#'s Linq library.

Code 6.5
```
# a custom PyESAPI collection wrapper
patient.CoursesLot()

## output ##
#  <pyesapi.Lot.Lot at 0x1b26c081a90>

# Indexable at construction
patient.CoursesLot(0).Id

## output ##
# 'C1'

# Indexable after construction
patientpatient.CoursesLot()[0].Id

## output ##
# 'C1'

# passing string will match Id field on objects
patient.CoursesLot('C1').Id

## output ##
# 'C1'

# selecting first occurrence or None
patient.CoursesLot().FirstOrDefault(lambda c: c.Id == 'C1').Id

## output ##
# 'C1'

# passing function at construction acts like .FirstOrDefault()
patient.CoursesLot(lambda c: c.Id == 'C1').Id

## output ##
# 'C1'
```

Open a Plan and Print Some Information

Code 6.6
```
plan = patient.CoursesLot('C1').PlanSetupsLot(0)
print(f'Plan Id: {plan.Id}')
print(f'Dose Per Fx: {plan.PrescribedDosePerFraction}')
print(f'Number of Fx: {plan.NumberOfFractions}')

## output ##
#   Plan Id: RA Calc
#   Dose Per Fx: 2.000 Gy
#   Number of Fx: 39

print(f'StructureID,TYPE,VOLUME')
for structure in plan.StructureSet.Structures:
    print(f'{structure.Id},{structure.DicomType},{structure.Volume:.2f
        }')

## output ##
#   StructureID,TYPE,VOLUME
#   External,EXTERNAL,39442.98
#   Bladder,ORGAN,387.96
#   CTV Prostate,CTV,35.39
#   z CouchInterior,SUPPORT,14789.42
#   FemoralHead_L,ORGAN,166.43
#   Bowel,ORGAN,589.58
#   PTV,PTV,100.97
#   FemoralHead_R,ORGAN,168.58
#   Rectum,ORGAN,69.59
#   z CouchSurface,SUPPORT,2723.04

print(f'BeamID,SSD,Mu,StartAngle')
for beam in plan.Beams:
    print(f'{beam.Id},{beam.SSD:.2f},{beam.Meterset.Value:.2f},'
          + f'{beam.ControlPoints[0].GantryAngle}')

## output ##
#   BeamID,SSD,Mu,StartAngle
#   ARC1-CW,886.17,326.31,181.0
#   ARC2-CCW,886.20,378.94,179.0
```

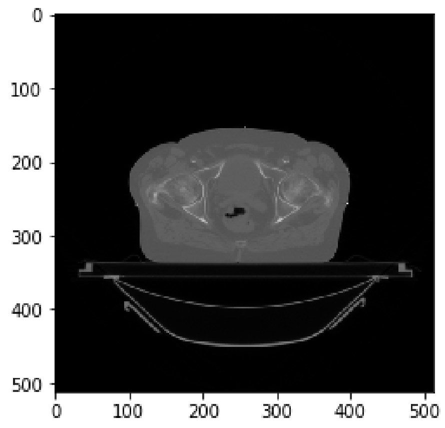
■

6.2.5 Plotting with Matplotlib

Plot a CT Slice

Code 6.7

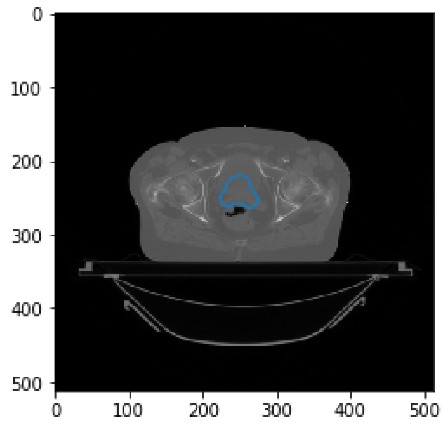
```
import matplotlib.pyplot as plt

ct = plan.StructureSet.Image  # just a shortcut
ct_image = plan.StructureSet.Image.np_array_like()
type(ct_image) # an actual numpy array!

## output ##
#   numpy.ndarray

print(ct.XSize, ct.YSize, ct.ZSize)
print(ct_image.shape)

## output ##
#   512 512 191
#   (512, 512, 191)

slice_num = 75
plt.imshow(ct_image[:,:,slice_num].T, cmap='gray')
plt.show()
```



Plot a CT Slice with a Contour

Code 6.8

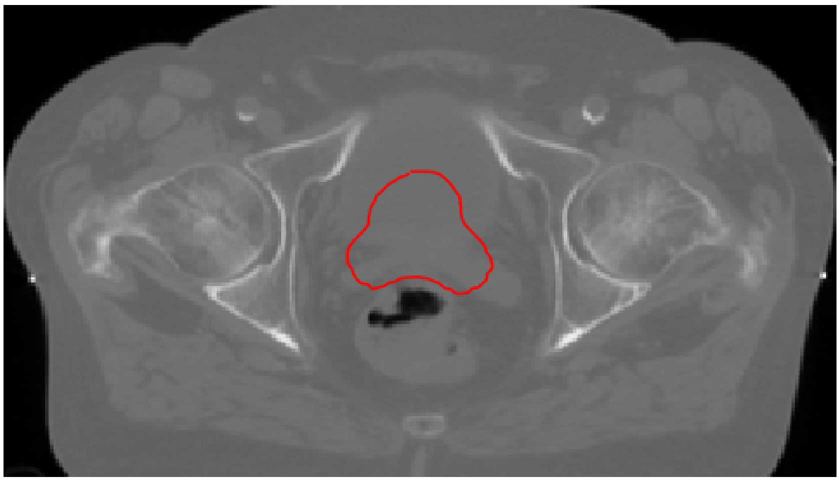
```
ptv_structure  = plan.StructureSet.StructuresLot('PTV')

plt.imshow(ct_image[:,:,slice_num].T, cmap='gray')
contours = ptv_structure.GetContoursOnImagePlane(slice_num)
for pt_list in contours:
    plt.plot(
        [(pt.x-ct.Origin.x)/ct.XRes for pt in pt_list],
        [(pt.y-ct.Origin.y)/ct.YRes for pt in pt_list]
    )
plt.show()
```



Crop and Zoom CT Plot

Code 6.9

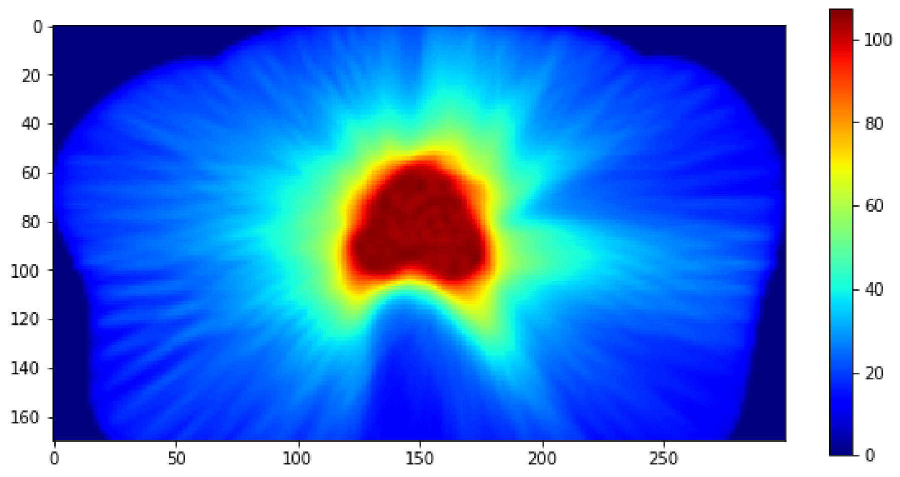
```
plt.figure(figsize=(10,5))
plt.imshow(ct_image[100:400,160:330,slice_num].T, cmap='gray')

for pt_list in ptv_structure.GetContoursOnImagePlane(slice_num):
    plt.plot(
        [(pt.x-ct.Origin.x)/ct.XRes - 100 for pt in pt_list],
        [(pt.y-ct.Origin.y)/ct.YRes - 160 for pt in pt_list],
        'r'
    )

plt.axis('off')  # turn of axis numbers
plt.show()
```



Plot Dose Distribution

Code 6.10

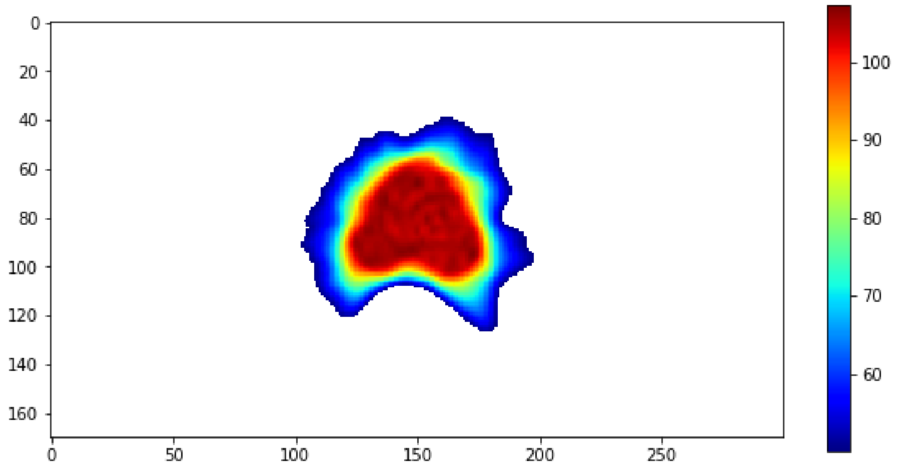
```
# creating dose at *CT resolution* takes about 50 seconds
dose = plan.Dose.np_array_like(plan.StructureSet.Image)
roi_slice=(slice(100,400),slice(160,330),slice_num)

plt.figure(figsize=(10,5))
plt.imshow(dose[roi_slice].T,cmap='jet')
plt.colorbar()
plt.show()
```



Plot Dose Distribution Above Threshold

Code 6.11

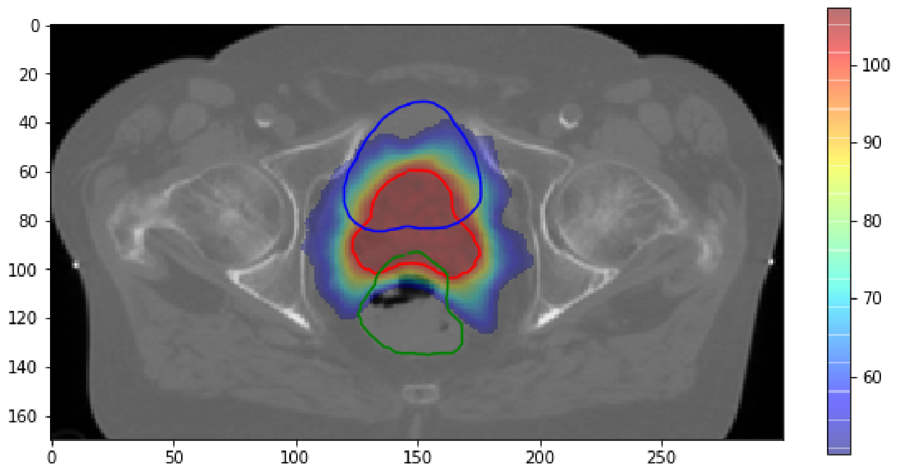
```
import numpy as np
dose_gt50 = np.ma.masked_where(dose<50.,dose)

plt.figure(figsize=(10,5))
plt.imshow(dose_gt50[roi_slice].T, cmap='jet')
plt.colorbar()
plt.show()
```



Plot CT with Contours and Dose Distribution Above Threshold

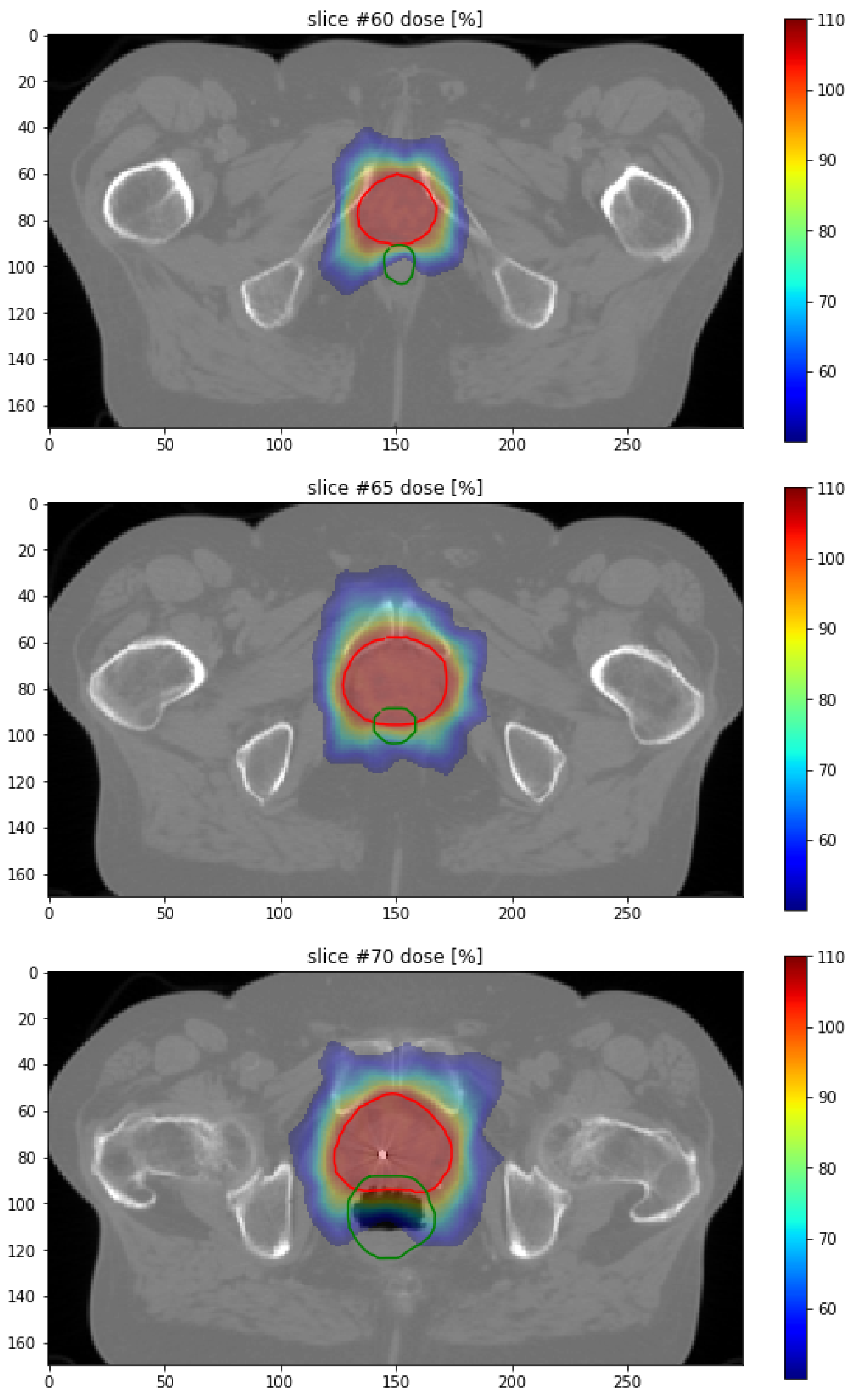
Code 6.12

```
def plot_contour_at_slice(structure, roi, style):
    for pt_list in structure.GetContoursOnImagePlane(roi[2]):
        plt.plot(
            [(pt.x-ct.Origin.x)/ct.XRes - roi[0].start for pt in pt_list
                ],
            [(pt.y-ct.Origin.y)/ct.YRes - roi[1].start for pt in pt_list
                ],
            style
        )


plt.figure(figsize=(10,5))
plt.imshow(ct_image[roi_slice].T, cmap='gray')
plt.imshow(dose_gt50[roi_slice].T, cmap='jet', alpha=.33)

plot_contour_at_slice(ptv_structure, roi_slice, 'r')
plot_contour_at_slice(plan.StructureSet.StructuresLot('Bladder'),
    roi_slice,'b')
plot_contour_at_slice(plan.StructureSet.StructuresLot('Rectum'),
    roi_slice, 'g')

plt.colorbar()
plt.show()
```



Plotting Multiple Slices

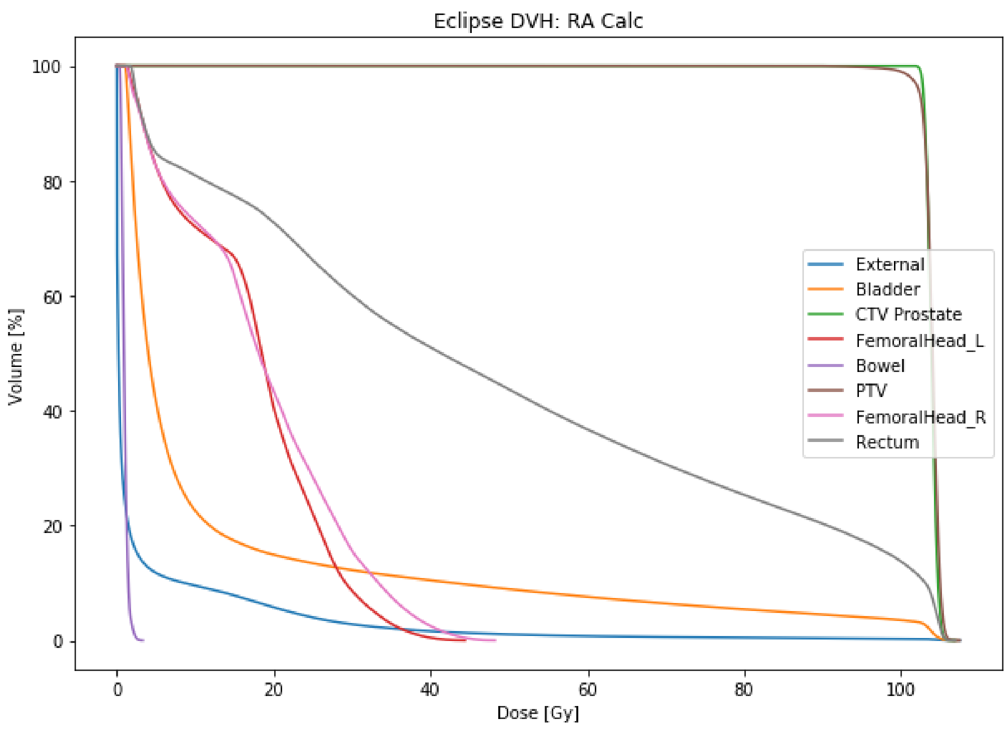
Code 6.13
```
def plot_slice(slice_index,dose_level):
    plt.figure(figsize=(10,5))
    plt.imshow(ct_image[roi_slice[0],roi_slice[1],slice_index].T, cmap='
        gray', vmax=2000)

    dose_slice = dose[roi_slice[0],roi_slice[1],slice_index]
    dose_slice_mask = np.ma.masked_where(dose_slice<dose_level,
        dose_slice)
    plt.imshow(dose_slice_mask.T, cmap='jet', alpha=.3, vmax=110)

    cb=plt.colorbar()
    cb.set_alpha(1.0)  # so colorbar is not transparent
    cb.draw_all()

    _roi = (roi_slice[0],roi_slice[1],slice_index)
    plot_contour_at_slice(ptv_structure,_roi,'r')
    plot_contour_at_slice(plan.StructureSet.StructuresLot('Bladder'),
        _roi,'b')
    plot_contour_at_slice(plan.StructureSet.StructuresLot('Rectum'),_roi
        ,'g')

    plt.title(f'slice #{slice_index} dose [%]')
    plt.show()


for i in range(60,75,5):
    plot_slice(i,50)
```



Plot DVH

Code 6.14

```
def plot_dvh(structure):
    dvh = plan.GetDVHCumulativeData(
        structure,
        pyesapi.DoseValuePresentation.Relative,
        pyesapi.VolumePresentation.Relative,
        .01
    )
    if dvh is not None:
        dose_x = [p.DoseValue.Dose for p in dvh.CurveData]
        volume_y = [p.Volume for p in dvh.CurveData]
        plt.plot(dose_x, volume_y, label=structure.Id)


plt.figure(figsize=(10,7))
for structure in plan.StructureSet.Structures:
    plot_dvh(structure)
plt.legend(loc=0)
plt.title(f'Eclipse DVH: {plan.Id}')
plt.ylabel('Volume [%]')
plt.xlabel('Dose [Gy]')
plt.show()
```



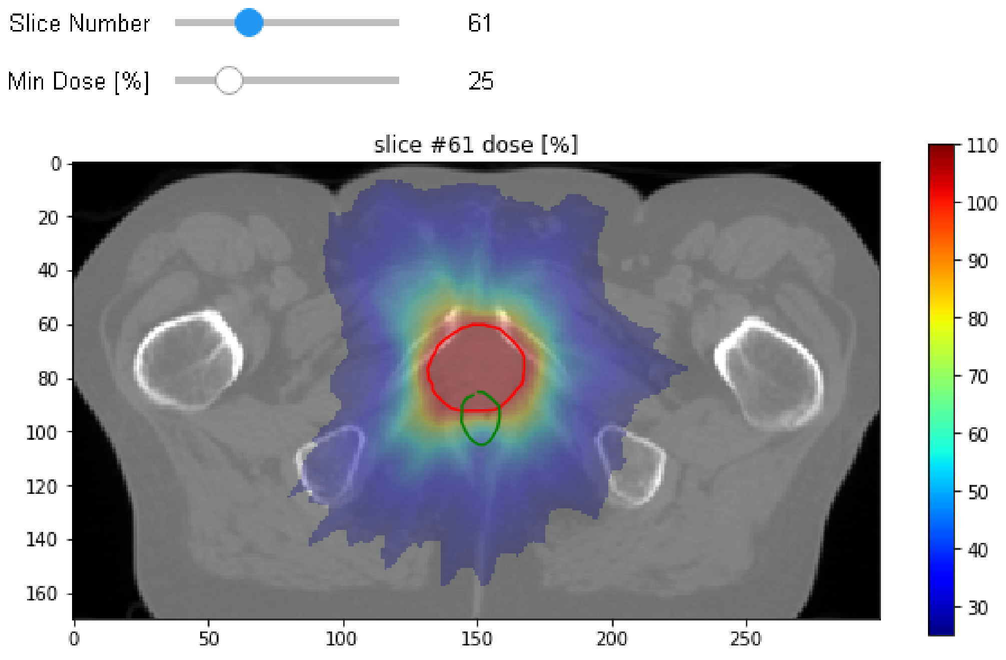
6.2.6 Interactive Plots

Interactive Slice Viewer

Code 6.15

```
from ipywidgets import interactive, widgets

slice_slider = widgets.IntSlider(
    value=slice_num,
    min=0,
    max=ct_image.shape[2]-1,
    step=1,
    description='Slice Number',
    continuous_update=False,
)
```

```
dose_slider = widgets.IntSlider(
    value=30,
    min=0,
    max=110,
    step=1,
    description='Min Dose [%]',
    continuous_update=False,
)

interactive(
    plot_slice,
    slice_index=slice_slider,
    dose_level=dose_slider
)
```

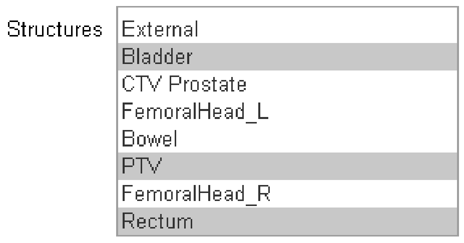

Slice Number 61

Min Dose [%] 25

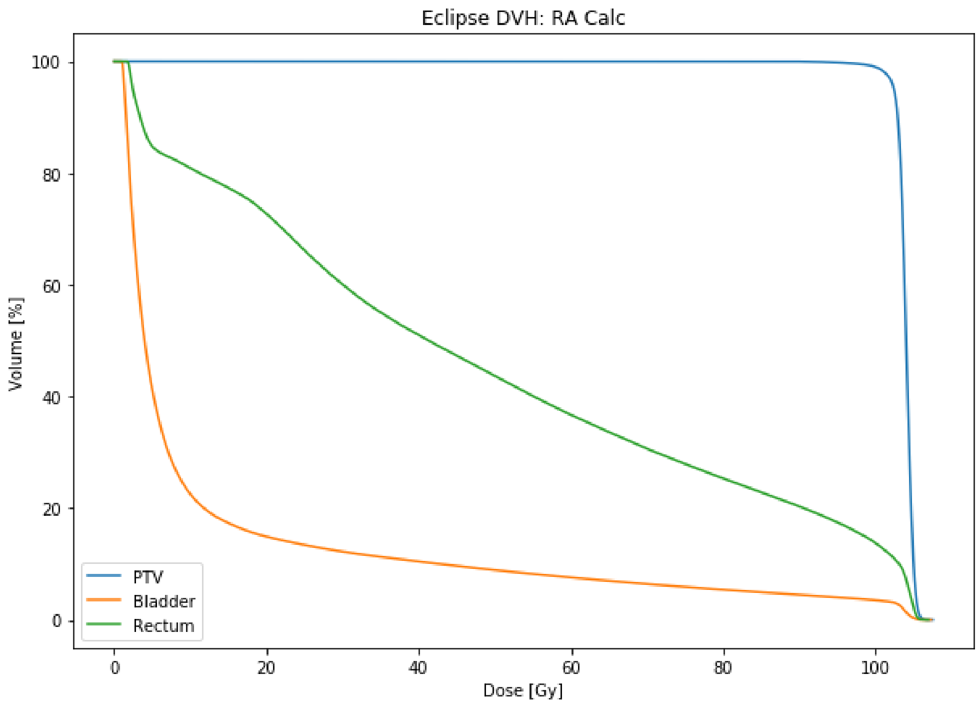
Interactive DVH Viewer

Code 6.16

```
\begin{lstlisting}
def plot_dvh_interactive(structure_ids):
    plt.figure(figsize=(10,7))
    for structure_id in structure_ids:
        structure = plan.StructureSet.StructuresLot(structure_id)
        plot_dvh(structure)
    plt.legend(loc=0)
    plt.title(f'Eclipse DVH: {plan.Id}')
    plt.ylabel('Volume [%]')
    plt.xlabel('Dose [Gy]')
    plt.show()

structure_id_choices = list(
    filter(lambda s: 'Couch' not in s.Id, plan.StructureSet.Structures)
)

structure_multi_select = widgets.SelectMultiple(
    options=[_.Id for _ in structure_id_choices],
    value=['PTV','Bladder','Rectum'],
    rows=len(structure_id_choices),
    description='Structures',
    disabled=False
)

interactive(
    plot_dvh_interactive,
    structure_ids=structure_multi_select
)
```



```
Structures  External
            Bladder
            CTV Prostate
            FemoralHead_L
            Bowel
            PTV
            FemoralHead_R
            Rectum
```



6.3 Data Mining

In this section a new notebook is created to demonstrate ways of extracting, storing, and saving data from Eclipse plans using PyESAPI. For discreet well-formed data, database tables (or even spreadsheet files) are adequate. For bulk/non-uniformly structured data like: DVH Curves, CT images, dose distributions, and control point sequences, an advanced scientific data container like HDF5 is much more appropriate. Both methods of data extraction, storage, and retrieval using Pandas, SQLite3, and h5py will be presented.

6.3.1 Extracting Data with Pandas

In Pandas, data tables are represented with the DataFrame object. This section has examples of how to use it. For more information, please lookup the official Pandas documentation.

DataFrame: Basic Structure Information

Code 6.17

```
# boilerplate
import pyesapi
import atexit
app = pyesapi.CustomScriptExecutable.CreateApplication('python_demo')
atexit.register(app.Dispose);

# pandas
import pandas as pd

# grab a patient and a plan
app.ClosePatient()  # good practice
a_patient = app.OpenPatientById('RapidPlan-01')
a_plan = a_patient.CoursesLot(0).PlanSetupsLot(0)

# create data-frame
df = pd.DataFrame(
    [(s.Id, s.DicomType, s.Volume, s.IsHighResolution) for s in a_plan.
        StructureSet.Structures],
    columns = ('StructureId', 'DicomType', 'Volume[cc]', 'IsHighRes')
)
print(df)

## output ##
#          StructureId DicomType   Volume[cc]  IsHighRes
#  0          External  EXTERNAL  39442.979501      False
#  1           Bladder     ORGAN    387.956598      False
#  2      CTV Prostate       CTV     35.393592      False
#  3  z CouchInterior   SUPPORT  14789.423747      False
#  4    FemoralHead_L     ORGAN    166.430112      False
#  5             Bowel     ORGAN    589.581268      False
#  6               PTV       PTV    100.968767      False
#  7    FemoralHead_R     ORGAN    168.577012      False
#  8            Rectum     ORGAN     69.587825      False
#  9    z CouchSurface   SUPPORT   2723.042663      False
```

DataFrame: Adding Dose at Volume information

Code 6.18

```
def d_at_v(plan,structure,volume):
    _dose=plan.GetDoseAtVolume(structure,volume,
                               pyesapi.VolumePresentation.Relative,
                               pyesapi.DoseValuePresentation.Absolute)
    return _dose.Dose

columns = (
    'PatientId',
    'PlanId',
    'StructureId',
    'DicomType',
    'Volume(cc)',
    'IsHighRes',
    'D95%(Gy)',
    'D25%(Gy)',
    'D50%(Gy)'
)

def get_structure_info(plan):
    return [(
        plan.StructureSet.Patient.Id,
        plan.Id,
        s.Id,
        s.DicomType,
        s.Volume,
        s.IsHighResolution,
        d_at_v(plan,s,95),
        d_at_v(plan,s,25),
        d_at_v(plan,s,50)
    ) for s in plan.StructureSet.Structures]

df = pd.DataFrame(get_structure_info(a_plan),columns=columns)
print(df)

## output ##
#        PatientId   PlanId        StructureId  DicomType    Volume(cc)  \
#   0  RapidPlan-01  RA Calc          External   EXTERNAL  39442.979501
#   1  RapidPlan-01  RA Calc           Bladder      ORGAN    387.956598
#   2  RapidPlan-01  RA Calc      CTV Prostate        CTV     35.393592
#   3  RapidPlan-01  RA Calc  z CouchInterior    SUPPORT  14789.423747
#   4  RapidPlan-01  RA Calc     FemoralHead_L      ORGAN    166.430112
#   5  RapidPlan-01  RA Calc             Bowel      ORGAN    589.581268
#   6  RapidPlan-01  RA Calc               PTV        PTV    100.968767
#   7  RapidPlan-01  RA Calc     FemoralHead_R      ORGAN    168.577012
#   8  RapidPlan-01  RA Calc            Rectum      ORGAN     69.587825
#   9  RapidPlan-01  RA Calc   z CouchSurface    SUPPORT   2723.042663
#
#      IsHighRes   D95%(Gy)   D25%(Gy)   D50%(Gy)
#   0      False   0.001006   0.747400   0.151894
#   1      False   1.056218   6.851331   3.119362
#   2      False  80.209159  81.334786  80.987743
#   3      False        NaN        NaN        NaN
#   4      False   1.888115  18.756412  14.393730
#   5      False   0.376215   0.919412   0.702133
#   6      False  79.862116  81.523899  81.113483
#   7      False   1.672847  20.485591  14.070829
#   8      False   1.924328  62.892221  32.191499
#   9      False        NaN        NaN        NaN
```

■

DataFrame: Bulk Extraction

Code 6.19

```
patient_id_list= [
    'RapidPlan-01',
    'RapidPlan-02',
    'RapidPlan-03',
    'RapidPlan-04',
    'RapidPlan-06',
    'Eclipse-01',
    'Eclipse-03',
    'Eclipse-04',
    'Eclipse 06',
    'Eclipse 07',
    'Eclipse 11'
]
dataframe_list = []
for patient_id in patient_id_list:
    print(f'Loading structure data from {patient_id} plans...\t\t',end
        ='\r')
    app.ClosePatient()
    patient = app.OpenPatientById(patient_id)
    for course in patient.Courses:
        for plan in course.PlanSetups:
            if plan.Dose is not None:
                dataframe_list.append(pd.DataFrame(get_structure_info(
                    plan),columns=columns))
print('Done!'+' '*80)

structure_dataframe = pd.concat(dataframe_list,ignore_index=True)
print(structure_dataframe)

## output ##
#         PatientId      PlanId      StructureId  DicomType     Volume(
    cc)  \
#  0    RapidPlan-01     RA Calc         External   EXTERNAL
    39442.979501
#  1    RapidPlan-01     RA Calc          Bladder      ORGAN
    387.956598
#  2    RapidPlan-01     RA Calc     CTV Prostate        CTV
    35.393592
#  3    RapidPlan-01     RA Calc   z CouchInterior    SUPPORT
    14789.423747
#  4    RapidPlan-01     RA Calc     FemoralHead_L      ORGAN
    166.430112
#  5    RapidPlan-01     RA Calc            Bowel      ORGAN
    589.581268
#  6    RapidPlan-01     RA Calc              PTV        PTV
    100.968767
#  7    RapidPlan-01     RA Calc     FemoralHead_R      ORGAN
    168.577012
#  8    RapidPlan-01     RA Calc           Rectum      ORGAN
    69.587825
#  9    RapidPlan-01     RA Calc    z CouchSurface    SUPPORT
    2723.042663
#  10   RapidPlan-02       Boost       NS_Ring_05  AVOIDANCE
    534.570193
#  ..            ...         ...              ...        ...
    ...
#  303    Eclipse 11    Post MCO          Lt Lens      ORGAN
    0.216589
#  304    Eclipse 11    Post MCO           Lt Eye      ORGAN
    8.857691
#  305    Eclipse 11    Post MCO   Rt Optic Nerve      ORGAN
    0.858122
#  306    Eclipse 11    Post MCO   Lt Optic Nerve      ORGAN
    0.619426
#  307    Eclipse 11    Post MCO          Patient   EXTERNAL
    3528.742710
#  308    Eclipse 11    Post MCO     Optic Chiasm      ORGAN
    2.437153
```

```
#   309     Eclipse 11     Post MCO        PRV BS+3mm       ORGAN
     51.738760
#   310     Eclipse 11     Post MCO    PRV Optics+3mm       ORGAN
     12.306030
#   311     Eclipse 11     Post MCO              PTV60
     120.098720
#   312     Eclipse 11     Post MCO          pituitary
     0.741262
#
#        IsHighRes    D95%(Gy)    D25%(Gy)    D50%(Gy)
#   0        False    0.001006    0.747400    0.151894
#   1        False    1.056218    6.851331    3.119362
#   2        False   80.209159   81.334786   80.987743
#   3        False         NaN         NaN         NaN
#   4        False    1.888115   18.756412   14.393730
#   5        False    0.376215    0.919412    0.702133
#   6        False   79.862116   81.523899   81.113483
#   7        False    1.672847   20.485591   14.070829
#   8        False    1.924328   62.892221   32.191499
#   9        False         NaN         NaN         NaN
#   10       False    0.102603    4.889706    0.471733
#   ..         ...         ...         ...         ...
#   303      False    1.871476    2.113649    2.035836
#   304      False    1.879416    2.717094    2.451896
#   305      False    6.422729   41.844208   23.076247
#   306      False    3.329274   22.857895   13.970566
#   307      False    0.695551   16.770237    6.988062
#   308      False   17.395122   36.585495   24.459408
#   309      False   18.223272   42.680298   34.251112
#   310      False    7.024586   42.171339   26.928773
#   311      False   56.487299   62.450297   61.841293
#   312      False   18.047796   23.273955   21.274643
#
#   [313 rows x 9 columns]
```

DataFrame: Saving to CSV file

It is not really recommended to use a comma to separate columns since structure ids may contain commas. It's much safer to use a delimiter that is now allowed in structure ids, like the forward-slash "/". Or, even better, you could sanitize text fields as you insert into the DataFrame. Also, if you plan on importing into Excel, do not use "ID" as the first column name.

Code 6.20

```
with open('./StructureData.csv','w') as f:
    f.write(structure_dataframe.to_csv(sep='/'))
```

6.3.2 Pandas and SQLite

In principle, any Python SQL connector should work. SQLite is used as an example for the sake of simplicity.

Saving DataFrame to SQLite Table

Code 6.21

```
import sqlite3
```

```
sql_connection = sqlite3.connect('big_data.db')
table_name = 'structure_data'
# 'replace' table if it exists
structure_dataframe.to_sql(table_name,sql_connection, if_exists='replace
    ')
```

6.3.3 SQL Query to DataFrame

Basic Query

Code 6.22

```
df = pd.read_sql_query(f'select * from {table_name} limit 10;',
    sql_connection)
print(df)

## output ##
#      index      PatientId    PlanId        StructureId  DicomType     Volume(
    cc)   \
#   0        0   RapidPlan-01   RA Calc          External   EXTERNAL
    39442.979501
#   1        1   RapidPlan-01   RA Calc           Bladder      ORGAN
    387.956598
#   2        2   RapidPlan-01   RA Calc      CTV Prostate        CTV
    35.393592
#   3        3   RapidPlan-01   RA Calc   z CouchInterior    SUPPORT
    14789.423747
#   4        4   RapidPlan-01   RA Calc     FemoralHead_L      ORGAN
    166.430112
#   5        5   RapidPlan-01   RA Calc             Bowel      ORGAN
    589.581268
#   6        6   RapidPlan-01   RA Calc               PTV        PTV
    100.968767
#   7        7   RapidPlan-01   RA Calc     FemoralHead_R      ORGAN
    168.577012
#   8        8   RapidPlan-01   RA Calc            Rectum      ORGAN
    69.587825
#   9        9   RapidPlan-01   RA Calc    z CouchSurface    SUPPORT
    2723.042663
#
#      IsHighRes    D95%(Gy)    D25%(Gy)    D50%(Gy)
#   0           0    0.001006    0.747400    0.151894
#   1           0    1.056218    6.851331    3.119362
#   2           0   80.209159   81.334786   80.987743
#   3           0         NaN         NaN         NaN
#   4           0    1.888115   18.756412   14.393730
#   5           0    0.376215    0.919412    0.702133
#   6           0   79.862116   81.523899   81.113483
#   7           0    1.672847   20.485591   14.070829
#   8           0    1.924328   62.892221   32.191499
#   9           0         NaN         NaN         NaN
```

Query all PTV Data

Code 6.23

```
ptv_df = pd.read_sql_query(
        f'select * from {table_name} where DicomType=="PTV" limit 10;',
         sql_connection
)
print(ptv_df)

## output ##
#      index      PatientId       PlanId  StructureId  DicomType    Volume(cc)
      \
#   0       6  RapidPlan-01      RA Calc          PTV        PTV    100.968767
#   1      11  RapidPlan-02        Boost   PTV_Phase1        PTV   1069.403754
#   2      12  RapidPlan-02        Boost         PTVp        PTV    126.873081
#   3      23  RapidPlan-02   Boost Calc   PTV_Phase1        PTV   1069.403754
#   4      24  RapidPlan-02   Boost Calc         PTVp        PTV    126.873081
#   5      34  RapidPlan-03     H&N Calc        PTV70        PTV     42.097341
#   6      35  RapidPlan-03     H&N Calc   PTV56 Eval        PTV    302.535430
#   7      36  RapidPlan-03     H&N Calc        PTV56        PTV    362.115936
#   8      37  RapidPlan-03     H&N Calc   PTV63 Eval        PTV    121.562603
#   9      38  RapidPlan-03     H&N Calc        PTV63        PTV    166.037148
#
#      IsHighRes   D95%(Gy)   D25%(Gy)   D50%(Gy)
#   0          0  79.862116  81.523899  81.113483
#   1          0   0.129401   1.088077   0.389651
#   2          0  18.843981  19.409143  19.267913
#   3          0   0.127324   0.971796   0.360545
#   4          0  18.790938  19.410317  19.255657
#   5          0  70.234095  73.203606  72.630875
#   6          0  56.214419  59.991000  58.946876
#   7          0  56.224372  60.947360  59.212884
#   8          0  63.726866  68.179323  66.144457
#   9          0  63.846298  71.408508  67.229297
```

PTV Data Statistics

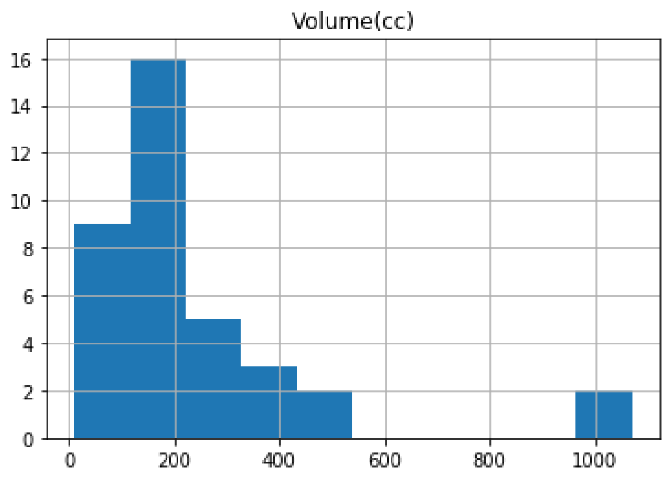
Code 6.24

```
ptv_df.mean()

## output ##
#   index          187.108108
#   Volume(cc)     239.577353
#   IsHighRes        0.000000
#   D95%(Gy)        51.154285
#   D25%(Gy)        54.176257
#   D50%(Gy)        53.250558
#   dtype: float64
```

6.3.4 Ploting With Pandas DataFrame

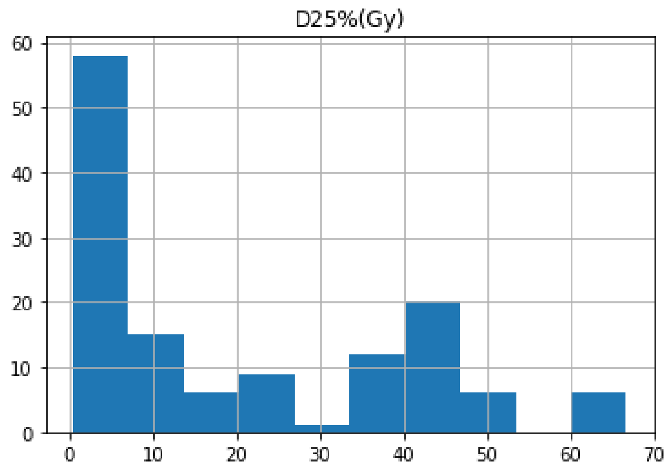
Plotting Distributions

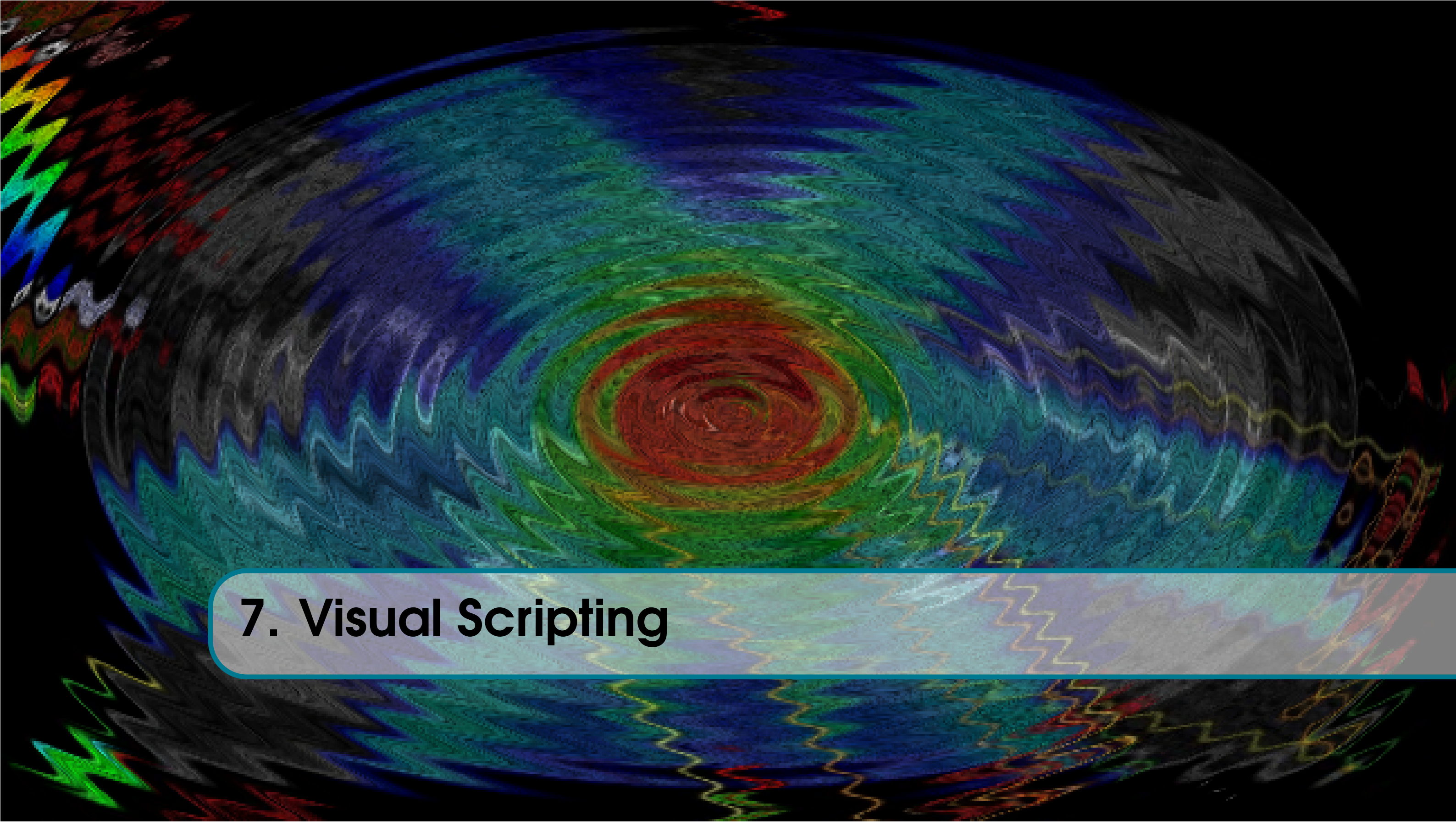
```
ptv_df.hist(column='Volume(cc)');
```



Code 6.25

```
pd.read_sql_query(
        f'select * from {table_name} where DicomType=="ORGAN";',
    sql_connection
).hist(column='D25%(Gy)');
```





7. Visual Scripting

MATTHEW SCHMIDT, MSc.

7.1 Introduction

The Visual Scripting Application within Eclipse Treatment Planning System (TPS) utilizes a concept known as Visual Programming to gather information from the currently active data model scope in a similar fashion to the Eclipse Scripting API. Visual Scripting comes with some inherent benefits such as streamlined methods to gather dose metrics and DVH information, built-in action packs for printing reports, writing to files, and capturing screenshots within the Eclipse Display view. The intended developers of Visual Scripting is a broad audience, from Eclipse users with limited programming experience to C# .NET developers that can write action packs to extend the capabilities of Visual Scripting.

Visual Scripting was introduced in Eclipse version 15.0, with as a standard feature accessible through the External Beam Planning workspace. It is a site license capable feature with license title *EclipseVisualScripting*. As with running scripts in read-only mode with the option *Approve Read-only scripts* unchecked in RT Administration: System and Facilities Workspace, there are no specific user rights needed to build, modify or run Visual Scripts. More information regarding the behavior of the *Approval Required for Read-only scripts* option will be available inside the **Visual Scripting Administration** section within this chapter.

This chapter aims to describe the basic concepts behind building Visual Scripts, the context items and action packs within the Visual Scripting Workbench, and the types of output achievable from Visual Scripts. Another section will target some of the subtleties in building custom action packs with the Eclipse Script Wizard. Finally, the administration of Visual Scripts will be discussed to detail the process behind sharing, storing Visual Scripts as favorites, and organizing these scripts.

7.2 Visual Scripting Basics

7.2.1 Visual Scripting Workbench

The Visual Scripting Workbench can be accessed from External Beam Planning. From the Tools menu, the selection *Visual Scripting* will open the following window [Figure 7.1].

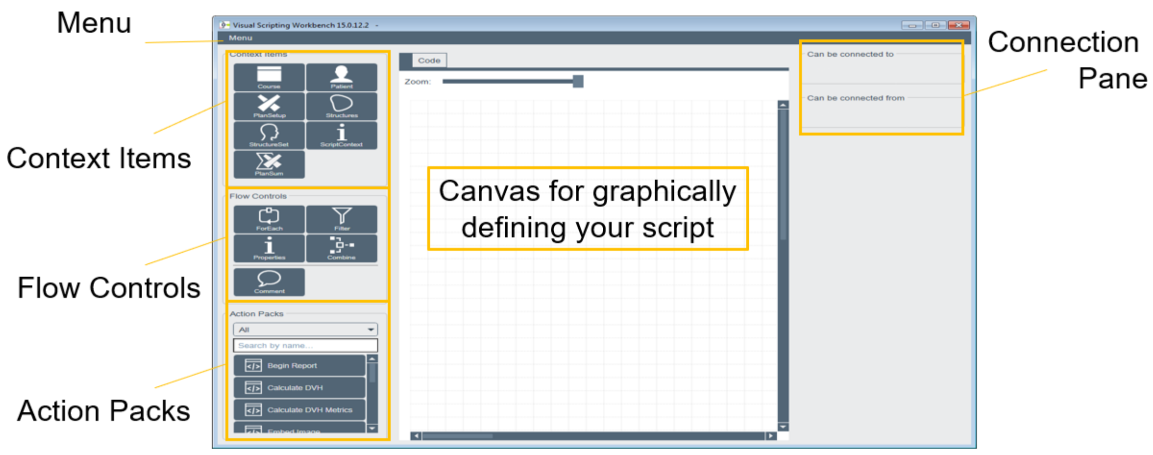


Figure 7.1: The Visual Scripting Workbench and its components.

The Menu pull-down within the Visual Scripting workspace, Figure 7.2, is the location by which the scripts will be saved and executed. There are 3 methods of saving scripts from the Menu.The save options for Visual Scripting include some important concepts that will be covered later within the chapter– import and export of Visual Scripts, loading custom action packs, and adding scripts to favorites.

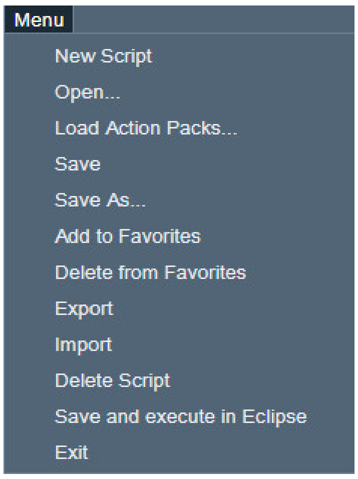


Figure 7.2: Items available from within the Visual Scripting Menu.

The context items within the Visual Scripting Workbench allows the developer access to the currently open data within the Eclipse Treatment Planning System. To use any context item, simply drag the context item into the canvas. Each context item will have some parameters for working with that code block. At the bottom right of each item is a small *i* that allows the developer to access the properties within that context item. The first context item to consider is the **ScriptContext** item [Figure 7.3]. From within these properties, further information about the data model object can be made available as either a new ESAPI class object or an enumeration of Eclipse class objects. In the example below, we can see that the properties of the ScriptContext context item include the Patient, Course, PlanSetup, StructureSet, PlanSums and Structures showing that the other context items can be considered as shortcuts from the ScriptContext. From each of the other Context Items, deeper level layers of properties are available for the specific object selected.

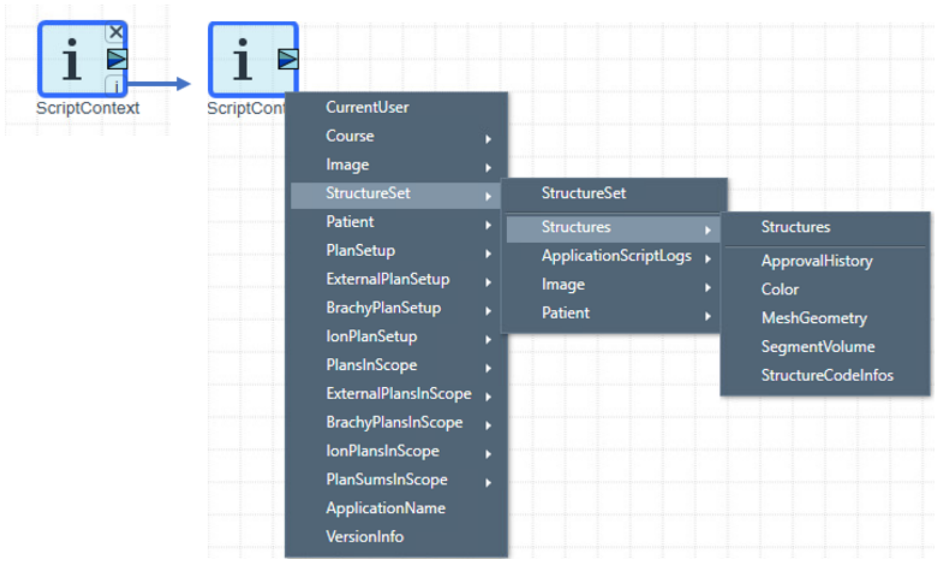


Figure 7.3: The ScriptContext context item and the properties within. Here the Structures of the current scope can be accessed from the ScriptContext item though there exists a Structures context item shortcut directly

The Flow Control section will allow for some common programming techniques to be implemented within the Visual Script. For example, it is common the programs to iterate through a collection of objects and select the object based on a characteristic of item. The Filter Flow Control can be employed in order implement this technique within Visual Scripting. Other flow controls can be used to combine data from various sources into a single entity or perform some action on data for each enumerated object within a collection. Examples of these flow controls will be utilized throughout the tutorial in the next section, **Building Visual Scripts**.

Action packs are the essential constituent to building effective and useful Visual Scripts. These are the methods with which calculations can be performed, PDF reports can be built, and screen shots within the planning workspace can be captured. As of version 15.5 of Visual Scripting, 11 action packs come installed for use in building Visual Scrifipts. Of those 11 action packs, 2 have the responsibility of calculating information directly from a Dose-Volume Histogram (DVH) while the other 9 deal with visualizing planning information for use in 3 output mechanisms: A pop-up display window (To View), comma-separated value (csv) file (To File), or PDF report (To Report). Visual Scripting is also extensible by the creation of custom action packs; this process can be followed in the section **Building Custom Action Packs**

7.2.2 Connecting Programming Blocks

When a programming block is dragged onto the canvas, there may be connections at the front or back of the item. Connections can be made manually between programming blocks by clicking into the outgoing arrow (on the right side) (Figure 7.4) of a block and dragging the mouse to the incoming port (on the left side) of another block. The Visual Scripting canvas will give immediate feedback on whether the connection that was initiated between the two programming blocks is valid, by color coding the connection as seen in Figure 7.5. An orange connection means that the connection between the two block elements is invalid, while blue and green colors represent a valid connection. Green color connections will occur whenever data being output is in the form of an IEnumerable (i.e. a DVH collection or table list of elements).

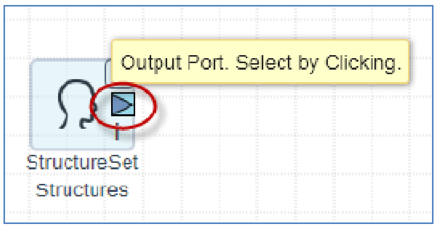


Figure 7.4: Each context item contains an output port to connect to other objects, while action packs contain input and output ports to connect to other elments.

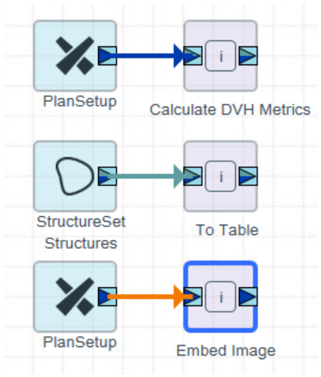


Figure 7.5: Programming block connections can be immediately validated based on the color returned by the connection. 3 types of connection colors can be seen here.

Programming blocks can also be connected by using the connection pane. The connection pane shows the script elements that can be used as inputs or outputs of the selected element. Figure 7.6.

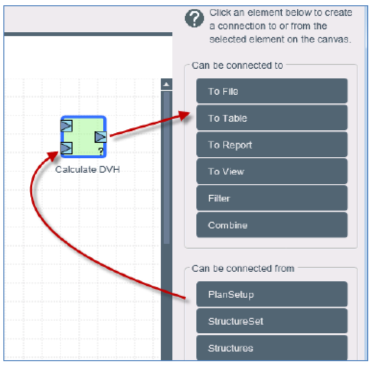


Figure 7.6: View Accepted inputs and outputs from the Connection Pane

7.3 Building Visual Scripts

This section covers the building of simple Visual Scripts within the Visual Scripting workbench. There are many types of Visual Scripts that one could build to evaluate a plan, generate reports, export dosimetric data and much more, but this section will organize the building of scripts into 3 categories: embedding reporting items, gathering dosimetric data, and tabular reporting. Throughout each of these categories a number of flow controls and action packs will be used to gain some insight into the use of these programming blocks.

7.3.1 Embedding Reporting Items

The report created below will be a simple report that only utilizes 4 distinct action packs. The purpose of this report is to combine an existing report with screenshots or images that may be of use to the planning documentation. A prerequisite for this report is that perhaps you have already saved a PDF of the patient documentation. In order to begin this report open the Visual Scripting Workbench and drag the action pack element **Begin Report** into the canvas. From the connection pane, click the following action pack items under the **Can be connected to** section: **Embed PDF** → **Embed Image** → **Embed Screenshot** → **Embed Screenshot** → **Embed Screenshot** → **Embed Screenshot** → **End Report**. At this point the canvas should look like 7.7

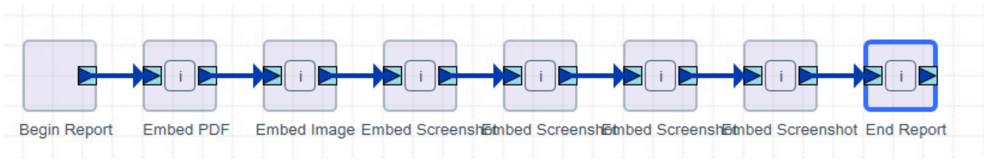
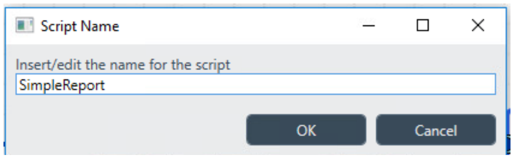


Figure 7.7: Initial Script to create report from existing data.

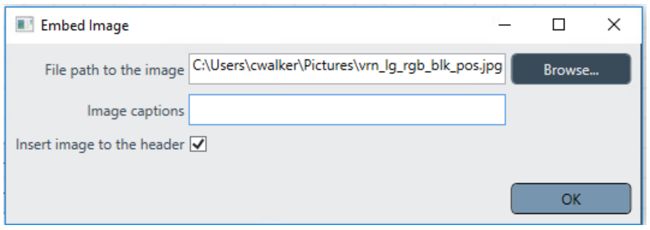
Now would be a suggested time to save the report by selecting the Menu and choosing the **Save** option: (Figure 7.8).



Insert/edit the name for the script
SimpleReport

Figure 7.8: Saving the progress of the initial Visual Script.

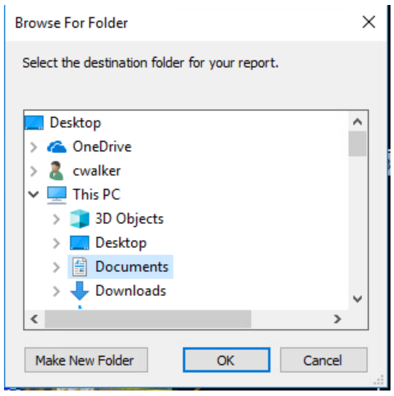
Select the icon in the center of the first **Embed Image** action pack. Upon selecting this icon, a popup window will allow for the input the location of the image that should be embedded and a caption for the image. For this reason, the image being embedded should be in a static location available to all users. This same icon will allow the developer to add a caption to the **Embed Screenshot** elements. For each of the four screenshot action packs, input a caption as the following text: *Transverse, Frontal, Sagittal, DVH*. These captions will be seen in the screenshot tool (Figure 7.12) and in the PDF report output. Figure 7.9.



File path to the image C:\Users\cwalker\Pictures\vrn_lg_rgb_blk_pos.jpg
Image captions
Insert image to the header ✓

Figure 7.9: Adding the location for the report to embed the image.

From the Menu option select **Save and Execute in Eclipse**. When the visual script initializes, a dialog box will appear prompting for the location with which to save the PDF that will be generated. (Figure 7.10).



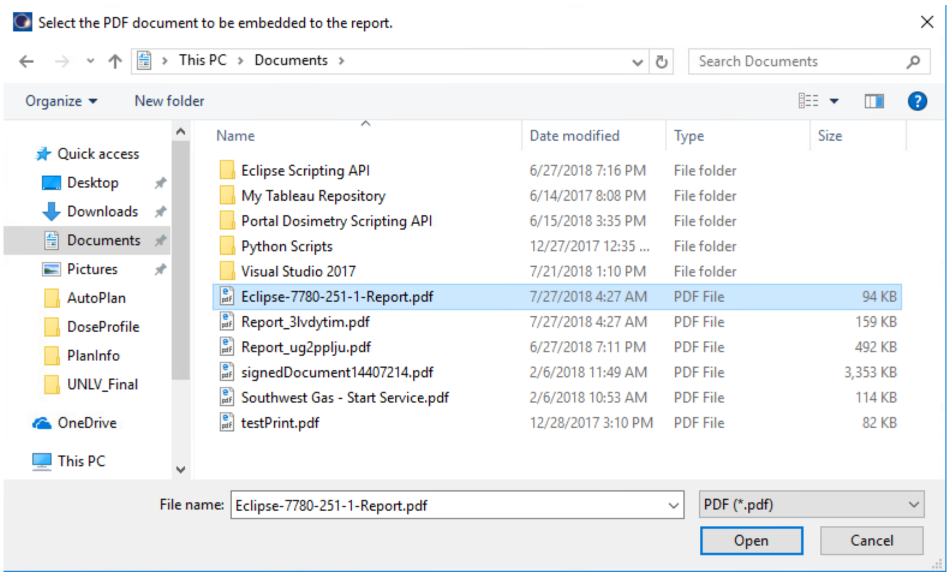
Browse For Folder

Select the destination folder for your report.

- Desktop
  - OneDrive
  - cwalker
  - This PC
    - 3D Objects
    - Desktop
    - Documents
    - Downloads

Figure 7.10: Select a location where the new PDF Report should be generated

Upon selecting the save selection location, another dialog box prompting the location with which to import and embed the PDF report. (Figure 7.11).



Select the PDF document to be embedded to the report.

This PC › Documents ›

Name	Date modified	Type	Size
Eclipse Scripting API	6/27/2018 7:16 PM	File folder	
My Tableau Repository	6/14/2017 8:08 PM	File folder	
Portal Dosimetry Scripting API	6/15/2018 3:35 PM	File folder	
Python Scripts	12/27/2017 12:35 ...	File folder	
Visual Studio 2017	7/21/2018 1:10 PM	File folder	
Eclipse-7780-251-1-Report.pdf	7/27/2018 4:27 AM	PDF File	94 KB
Report_3lvdytim.pdf	7/27/2018 4:27 AM	PDF File	159 KB
Report_ug2pplju.pdf	6/27/2018 7:11 PM	PDF File	492 KB
signedDocument14407214.pdf	2/6/2018 11:49 AM	PDF File	3,353 KB
Southwest Gas - Start Service.pdf	2/6/2018 10:53 AM	PDF File	114 KB
testPrint.pdf	12/28/2017 3:10 PM	PDF File	82 KB

File name: Eclipse-7780-251-1-Report.pdf

PDF (*.pdf)

Figure 7.11: Select the location of the PDF that should be embedded into the report

Finally, the Embed Screenshot tool will appear. The caption of the tool reminds the user running the Visual Script of how to prepare the Display View to acquire this screenshot. In this example, maximize the Frontal View and prepare the isodose lines, zoom, pan, and other image setting for the capturing of this screenshot. (Figure 7.12).

The report should be generated in the location determined in Figure 7.11. The report will also open with the computers default PDF viewer. Notice that the PDF and images are embedded at the end of the report with bookmarked links to navigate quickly to those items. The screenshots can also be seen in the report. Sit back and admire this easily generated PDF.

7.3.2 Gathering Dosimetric Data

In the Visual Scripting Workbench, select **New Script** from the Menu dropdown. This will clear the canvas of all programming blocks. In this example, dosimetric data will be gathered with two

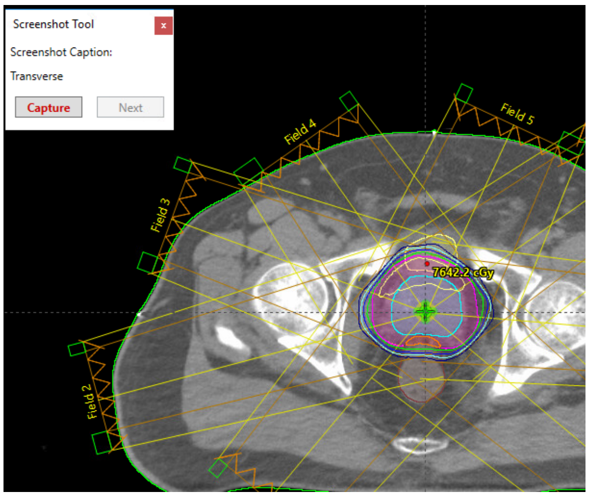


Figure 7.12: Prepare the Display View with the image settings to be captured by the screenshot tool.

action packs: **Calculate DVH** and **Calculate DVH Metrics**. This section will also explore two additional output action packs, **To View** and **To File**.

To begin with the new report, drag the **Calculate DVH** action pack into the canvas. From the flow controls, notice the available selections underneath **Can be connected from** where its evident that this action pack requires some input of PlanSetup(s) and Structures (StructureSet). Select the context items **PlanSetup** and **Structures** from the connection pane to connect them behind the **Calculate DVH** action pack. Under the **Can be connected to** section from the connection pane, click on the **To View** action pack. At this time, the canvas should look as in Figure 7.13

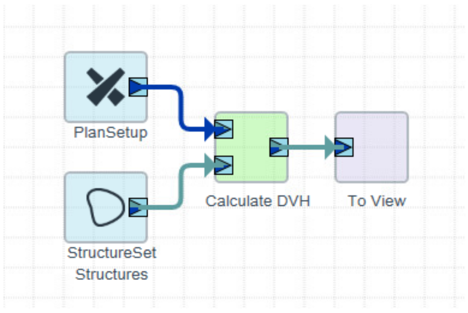
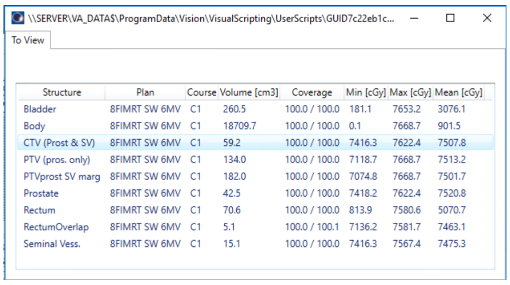


Figure 7.13: Canvas View for calculating the DVH statistics and printing them to view.

From the Menu item select **Save and Execute in Eclipse** to see the output from the Visual Script. The Calculate DVH action pack will show the standard dose statistics of each structure's DVH Data (Figure 7.14). The individual curve data is available as well from this action pack, but must be extracted from the **To Table** action pack that will be covered later in this chapter.

At this time, it may be conducive to mention one of the flow controls, **Filter**. Break the connection between the Structures context item and the Calculate DVH action pack by clicking on the "X" on the connection. This "X" becomes available when hovering over the connection line seen in Figure 7.15. Drag in the **Filter** flow control into the canvas and manual connect the Structure action pack behind it.

The **Filter** flow control works as follows: The first drop-down in the control is looking to the properties of the previous element connected to it. Once a property is selected, if the property is a string object, C# string comparison operators such as *StartsWith*, *EndsWith*, *Equals* and *Contains*


\\SERVER\VA_DATA$\ProgramData\Vision\VisualScripting\UserScripts\GUID7c22eb1c...

To View

Structure	Plan	Course	Volume [cm3]	Coverage	Min [cGy]	Max [cGy]	Mean [cGy]
Bladder	8FIMRT SW 6MV	C1	260.5	100.0 / 100.0	181.1	7653.2	3076.1
Body	8FIMRT SW 6MV	C1	18709.7	100.0 / 100.0	0.1	7668.7	901.5
CTV (Prost & SV)	8FIMRT SW 6MV	C1	59.2	100.0 / 100.0	7416.3	7622.4	7507.8
PTV (pros. only)	8FIMRT SW 6MV	C1	134.0	100.0 / 100.0	7118.7	7668.7	7513.2
PTVprost SV marg	8FIMRT SW 6MV	C1	182.0	100.0 / 100.0	7074.8	7668.7	7501.7
Prostate	8FIMRT SW 6MV	C1	42.5	100.0 / 100.0	7418.2	7622.4	7520.8
Rectum	8FIMRT SW 6MV	C1	70.6	100.0 / 100.0	813.9	7580.6	5070.7
RectumOverlap	8FIMRT SW 6MV	C1	5.1	100.0 / 100.1	7136.2	7581.7	7463.1
Seminal Vess.	8FIMRT SW 6MV	C1	15.1	100.0 / 100.0	7416.3	7567.4	7475.3

Figure 7.14: To View action pack showing results directly to the user running the Visual Script

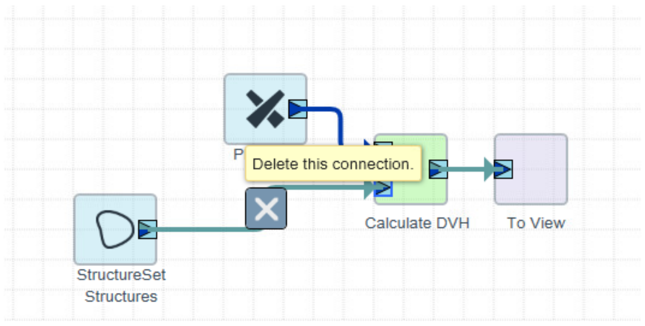


Figure 7.15: Connections can be deleted

will become available in the second drop-down. If the property selected is a boolean type, the operators available are *true* and *false*.

With this Filter, select the **DicomType** property, **Equals** and **CTV** for the 3 input options available. Repeat this step by dragging in another Structures context item and, connecting to another Filter flow control, but type **PTV** in the last text box. At this point the canvas should look like Figure 7.16.

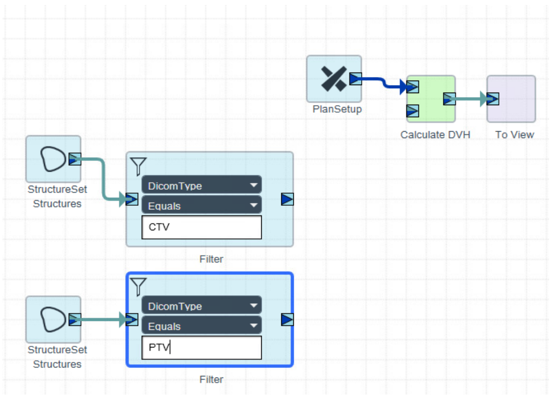


Figure 7.16: Two Filter flow control items filtering the structure items into PTV and CTV DICOM Types.

The **Combine** flow control item will allow for multiple fields of the same element value output type. This will allow the developer to apply values from multiple datasets into the same output. Connect both Filter flow control elements to the same Combine element, and connect the output of the Combine element to the Calculate DVH action pack. At this point the canvas should look like Figure 7.17. At this point, notice that the output only displays DVH data for structures of type *CTV* and *PTV* (Figure 7.18).

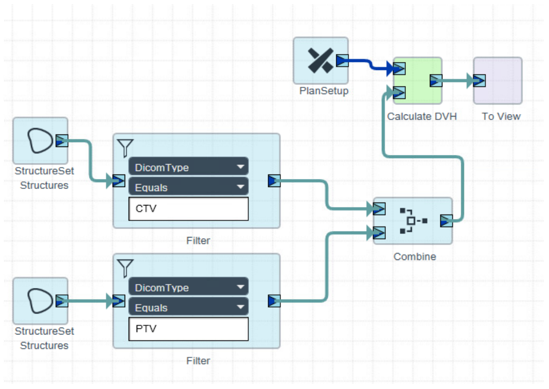
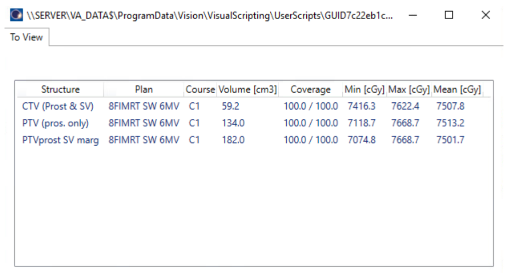


Figure 7.17: Apply the Combine flow control to connect two datasets of similar types into a single output.


\\SERVER\VA_DATA$\ProgramData\Vision\VisualScripting\UserScripts\GUID7c22eb1c...

To View

Structure	Plan	Course	Volume [cm3]	Coverage	Min [cGy]	Max [cGy]	Mean [cGy]
CTV (Prost & SV)	8FIMRT SW 6MV	C1	59.2	100.0 / 100.0	7416.3	7622.4	7507.8
PTV (pros. only)	8FIMRT SW 6MV	C1	134.0	100.0 / 100.0	7118.7	7668.7	7513.2
PTVprost SV marg	8FIMRT SW 6MV	C1	182.0	100.0 / 100.0	7074.8	7668.7	7501.7

Figure 7.18: The view from filtering the Structure Set into only CTV and PTV DICOM types.

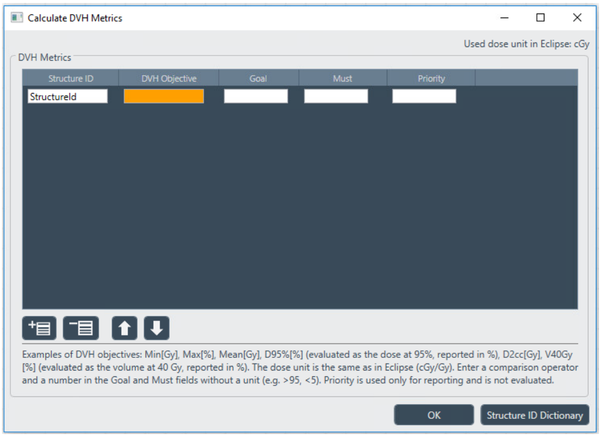
7.3.3 Calculation of Dose Quality Metrics

Another action pack suitable for calculation of more customized dose quality metrics is the **Calculate DVH Metric** action pack. Note the details of the DVH Metrics action pack include the Structure ID, the DVH Metric, Must and Goal levels and a priority (Figure 7.19).

The first column, Structure ID, allows for the input of the structure to which the metric should be applied. Here Structure Id can be typed in exactly as it is in Eclipse, or structures can be defined in the **Structure ID Dictionary** (Figure 7.20). This dictionary works left to right. First, input the name that should be called in the Structure ID column of the DVH Metrics table in the text box labeled **Add new structureID**. Then with that newly added structure ID selected in the list box (upper left), begin to add new alias IDs on the right side of the user interface. This will allow the alias IDs in the plan to be used in place of the structure ID.

Back in the DVH Metrics table, create a few rows to begin putting in some DVH Objectives. Add either a structure ID from the Structure Set or the Structure ID Dictionary to each of the first columns in each row. When writing the DVH objective, the first words can be **Min**, **Max**, or **Mean** or can represent a dose at a given volume or a volume at a given dose with **Dxx** or **Vyy**, respectively with ’xx’ being a given volume level and yy being a given isodose value. If a custom dose or volume value is requested, after xx and yy should come the input presentation of this value to be evaluated. This presentation could be in absolute units (cc for volume, cGy or Gy for dose depending on the dose unit in Eclipse– Seen in the upper right of this UI) or relative unit (%). Lastly, in square brackets, the output unit will need to be determined. These units will be the same for relative and absolute doses and volumes.

The Goal and Must columns can have Less Than (<) or Greater Than (>) signs followed by a numerical value for visual evaluation only. It is important to note that this Visual Script will not evaluate whether the DVH metric has met the user specified goal, but rather it is the responsibility



Used dose unit in Eclipse: cGy

DVH Metrics

Structure ID	DVH Objective	Goal	Must	Priority
StructureId				

Examples of DVH objectives: Min[Gy], Max[%], Mean[Gy], D95%[%] (evaluated as the dose at 95%, reported in %), D2cc[Gy], V40Gy [%] (evaluated as the volume at 40 Gy, reported in %). The dose unit is the same as in Eclipse (cGy/Gy). Enter a comparison operator and a number in the Goal and Must fields without a unit (e.g. >95, <5). Priority is used only for reporting and is not evaluated.

Figure 7.19: Columns of Calculating the DVH metrics. Note the Add Row, Delete Row, and Move row buttons at the bottom of the table.

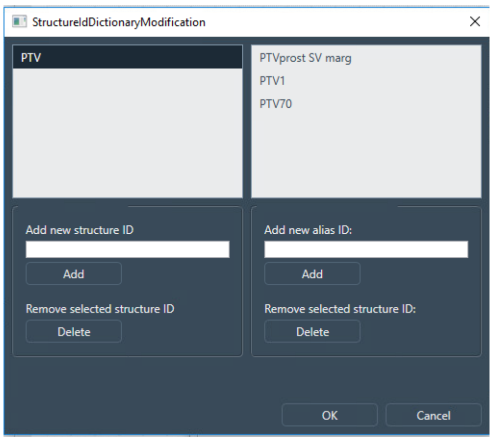


Figure 7.20: Creating Alias structure IDs with the StructureIdDictionaryModification tool

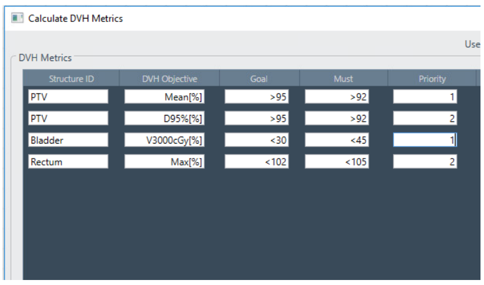
of the user reviewing theses metrics to make the pass/fail determination. Lastly the priority will be specific to clinical determination as well. The priority will be an integer value where the value of the integer could be identified with a severity of consequence for meeting that specific metric. For instance: (1 = Not severe; 2 = investigation needed; 3 = plan not of proper quality, replan needed). An example of such a completed table can be seen in Figure 7.21.

7.4 Building Action Packs

Probably the most powerful aspect to Visual Scripting is the ability to build custom action packs to extend the possibilities of what the visual script can accomplish. In this section we will look at the various components of Visual Scripting Action Packs along with a couple of examples to return ESAPI class enumerations and custom class enumerations.

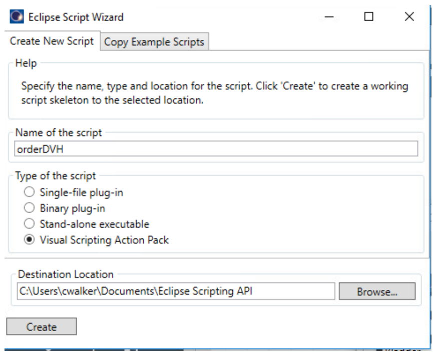
7.4.1 Components of Visual Scripting Action Packs

Visual Scripting action packs are more involved than their single-file plug-in, binary plug-in, and stand-alone executable counter-parts. Therefore, it is highly suggested to utilize the Eclipse Script Wizard in the creation of custom action packs. To begin, open the Eclipse Script Wizard, name the action pack (i.e. orderDVH), select **Visual Scripting Action Pack** and save to a location that will be remembered (Figure 7.22).



Structure ID	DVH Objective	Goal	Must	Priority
PTV	Mean[%]	>95	>92	1
PTV	D95%[%]	>95	>92	2
Bladder	V3000cGy[%]	<30	<45	1
Rectum	Max[%]	<102	<105	2

Figure 7.21: An example of a completed DVH Metric Table.



Help

Specify the name, type and location for the script. Click 'Create' to create a working script skeleton to the selected location.

Name of the script: orderDVH

Type of the script
- ○ Single-file plug-in
- ○ Binary plug-in
- ○ Stand-alone executable
- ● Visual Scripting Action Pack

Destination Location: C:\Users\cwalker\Documents\Eclipse Scripting API

Figure 7.22: An example of a completed DVH Metric Table.

Inside the project that is created by the Eclipse Script Wizard and looking inside of the c-sharp source code file (orderDVH.cs), there are a few things that should stand out in the code. Around line 12 the following comment and coding line occurs:

Code 7.1
```
\\TODO: Replace the following version attributes by creating
    AssemblyInfo.cs. You can do this in the properties of the Visual
    Studio project.
[assembly: AssemblyVersion("1.0.0.1")]
```

This line of code will only need to be changed if the checkbox **Approval Required for Read Only Scripts** is checked in the System Properties Tab of the System and Facilities workspace of RT Administration. If this checkbox is checked, more will be required of the TPS Administrator for running Visual Scripts. A more inclusive list can be found in the Visual Scripting Administration section.

The next coding section of interest starts on line 19.

Code 7.2
```
    \\TODO: Replace the existing class name with your own class name.
public class YourActionPackElement : VisualScriptElement
{
    public YourActionPackElement(){}
    public YourActionPackElement(IVisualScriptElementRuntimeHost host){}
}
```

Having an action pack with the class name YourActionPackElement is not going to cause an issue with running the action pack being created. The problem arises when two action packs target the same class name. If two or more action packs target the same class name, the visual script window will crash before the visual script can even be open. For this reason, it is best practice to give each custom action pack a unique class name.

At line 37, the following set of code allows the developer to modify the name that will be displayed to the user in Visual Scripting to the action pack.

Code 7.3
```
public override string DisplayName
{
    get
    {
        //TODO: Replace "Element␣Name" with the name you want displayed
           in the Visual Scripting UI.
        return "Element␣Name";
    }
}
```
■

Underneath this piece of code, another important interactive feature of Action Packs await. The following code allows for the user to change behavior of a certain *key* feature by setting the option to a certain *value* parameter in a key-value pair.

Code 7.4
```
IDictionary<string,string> m_options = new Dictionary<string,
    string>();
public override void SetOptions(string key, string value)
{
    m_options.Add(key,value);
}
public override IEnumerable<KeyValuePair<string, IEnumerable<string>>>
    AllowedOptions
{
    get
    {
        return new KeyValuePair<string, IEnumerable<string>>[]{
            new KeyValuePair<string, IEnumerable<string>>("TestOption",
                new string[]{"Test␣Value"})
        };
    }
}
```
■

In the code above, the only thing that needs to be changed is for each variable option the user can modify, a new Key-Value Pair should be created where *TestOption* is replaced by the name of the option and *Test Value* is replaced with the available options in the to that selection as a string array.

Lastly, line 31 is where the logic of the code is input. The syntax has the return object as the second word in line 31, while the input object type into the **Execute** method is in parenthesis. The code is written as below:

Code 7.5
```
public PlanSetup Execute(PlanSetup ps)
{
    // TODO: Add you code here.
    return null;
}
```
■

Note that in this piece of code, by default the Wizard has set the developer to input a PlanSetup object and returns to the visual script a PlanSetup class object. Also, it's important to remove the return null, as the return null will cause the output table to be empty rows of data if left.

7.4.2 Building the First Action Pack

The Action Pack that is about to be built, is meant to show an example of the use of the code pieces of the Visual Scripting Action Pack. It will input a ESAPI class object, or an enumeration of such, and output a similar data type. First, find the proper settings for the action pack being built. Change YourActionPackElement class name on line 20– remember this class name must be unique to other class names used in the visual script.

Code 7.6
```
public class orderDVHPackElement : VisualScriptElement
{
    public orderDVHPackElement() {}
    public orderDVHPackElement(IVisualScriptElementRuntimeHost host) {}
}
```

Next, change the DisplayName property to renname the action pack.

Code 7.7
```
public override string DisplayName
{
    get
    {
        //TODO: Replace "Element␣Name" with the name that you want to be
             displayed in the Visual Scripting UI.
        return "Order␣DVH";
    }
}
```

To see how the Visual Scripting UI will handle the user defined options, change the AllowedOptions property to let hte user select whether they want the DVH list in ascending or descending order as well as choose whether to sort by Min, Max or Mean doses as follows:

Code 7.8
```
public override IEnumerable<string, IEnumerable<string>>>
    AllowedOptions
{
    get
    {
        return new KeyValuePair<string, IEnumerable<string>>[]
        {
            new KeyValuePair<string, IEnumerable<string>>("Order", new
                string[] {"Ascending", "Descending"}),
            new KeyValuePair<string, IEnumerable<string>>{"Metric", new
                string[] {"Max","Mean","Min"})
            };
        }
    }
}
```

As stated in the above section, the last piece of code to modify is the Execute method. Since this action pack is to take an enumeration of DVHData class objects as input and return the same dat as output, both input and output data types must change to IEnumerable<DVHData>. Then, sort and order the data according to user-defined properties.

Code 7.9
```
public IEnumerable<DVHData> Execute(IEnumerable<DVHDat> dvhs)
{
    // TODO: Add your code here.
    List<DVHData> dvh_list = new List<DVHData>();
    //sort by user-defined metric.
    switch (m_options["Metric"])
    {
        case "Max":
            dvh_list = dvhs.OrderBy(x=>x.MaxDose.Dose).ToList();
            break;
        case "Mean":
            dvh_list = dvhs.OrderBy(x=>x.MeanDose.Dose).ToList();
            break;
        case "Min":
            dvh_list = dvhs.OrderBy(x=>x.MinDose.Dose).ToList();
            break;
        }
        if(m_options["Order"] == "Descending")
        {
            dvh_list.Reverse();
        }
        return dvh_list;
    }
}
```

7.4.3 Running a Visual Script

Loading an action pack into a visual script is a multi-step process. First, the action pack needs to be compiled. This action can be performed by selecting **Build Solution** from the **Build** menu of Visual Studio. This will cause the file with the extension *.vs.dll to be built in either the Debug or Release folder in the project (depending on the Visual Studio setting).

Locate this folder, and the file with the [projectname].vs.dll, and copy the file. Next find the following location in the File Data Server: `//<servername>/va_data$/ProgramData/Vision/VisualScripting/CustomActionPacks`

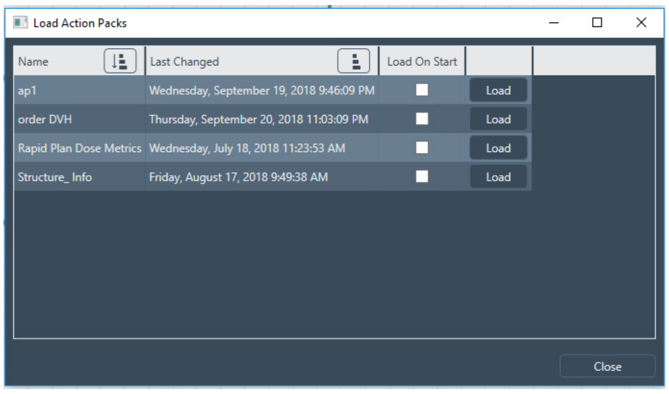
The name of the server that hosts the file data server can be found in the Varian Service Portal. Paste the custom action pack file here.

Back in the Visual Scripting UI, open the script from Figure 7.17. In order to load the custom action pack into the UI select **Menu**, **Load Action Packs**. Then the window in Figure 7.23 will allow the action pack to be loaded into the UI location for Action Packs by clicking on the Load button next to the action pack of interest.

Drop the newly available Order DVH action pack between the Calculate DVH action pack and the To View action pack. Note you can perform this drop by grabbing Order DVH from the action pack list or selecting Calculate DVH and choosing the Order DVH action pack from the connection pane. Note that in the settings from the AllowedOptions property can be set by clicking on the *i* in the center of the Action Pack. (Figure 7.24) Note that the DVH list comes in the order expressed by the options within the action pack.

7.4.4 Custom Action Pack with Custom Class Enumeration Output

Creating a custom action pack does not have to be limited to returning an ESAPI class object. Instead, one can choose to generate their own class and return. In this section, this will be shown while creating a simple plan checker. First, use the Eclipse Script Wizard to create a new Visual


Load Action Packs

Name	Last Changed	Load On Start	
ap1	Wednesday, September 19, 2018 9:46:09 PM	☐	Load
order DVH	Thursday, September 20, 2018 11:03:09 PM	☐	Load
Rapid Plan Dose Metrics	Wednesday, July 18, 2018 11:23:53 AM	☐	Load
Structure_ Info	Friday, August 17, 2018 9:49:38 AM	☐	Load

Close

Figure 7.23: An example of a completed DVH Metric Table.

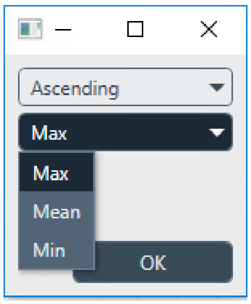


Figure 7.24: An example of a completed DVH Metric Table.

Scripting Action Pack. Name this project **planCheck**.

To build this action pack remember to change the class name (e.g. planCheckPackElement), the DisplayName (e.g. PlanCheck 9000). Also, give this action pack some options (e.g. "Prostate", "Head and Neck", "Lung" for a "Site" options and "IMRT", "VMAT","3DCRT" for a "Modality" option).

Next Create the custom class inside the planCheckPackElement class but outside of any of the properties. The custom class may be styled as such.

Code 7.10
```
public class PCheck
{
    public string Check {get;set;}
    public string Evaluation {get;set;}
    public string Result{get;set;}
}
```

Next in the Execute method, leave the input as PlanSetup, but set the output to an IEnumerable<PCheck> type.

Code 7.11
```
public IEnumerable<PCheck> Execute(PlanSetup ps)
        {
            List<PCheck> checks = new List<PCheck>();
            // TODO: Add your code here.
            //first create a plan check to make sure the target exists.
            PCheck pc1 = new PCheck();
            pc1.Check = "Target␣Defined";
            bool target = !String.IsNullOrEmpty(ps.TargetVolumeID);
            pc1.Evaluation = target ? $"Target␣Volume␣{ps.TargetVolumeID
                }" : "No␣Target␣Volume␣ID";
            pc1.Result = target ? "Pass" : "Fail";
            checks.Add(pc1);
            //next check if the max dose is inside the target volume.
            PCheck pc2 = new PCheck();
            pc2.Check = "Max␣Dose␣inside␣target";
            VVector max_loc = ps.Dose.DoseMax3DLocation;
            DoseValue dv = ps.Dose.DoseMax3D;
            Structure targetVolume = ps.StructureSet.Structures.First(x
                => x.Id == ps.TargetVolumeID);
            pc2.Evaluation = targetVolume.IsPointInsideSegment(max_loc)
                ?
                $"{dv}␣is␣inside␣{targetVolume.Id}" : $"{dv}␣is␣not␣
                    inside␣{targetVolume.Id}";
            pc2.Result = targetVolume.IsPointInsideSegment(max_loc) ? "
                Pass" : "Fail";
            checks.Add(pc2);
            //another check to see if plan dose was re-normalized after
                Leaf Motion calculation
            if(m_options["Modality"] == "IMRT")
            {
                //this check should be for IMRT only.
                double mu = ps.Beams.First().Meterset.Value;//get hte
                    first beams MU.
                //calculate the first beams MU from LMC.
                double dmax = 0;
                double fmu = 0;
                foreach(string line in ps.Beams.First().CalculationLogs.
                    First(x=>x.Category == "LMC").MessageLines)
                {
                    //note this doesn't apply for large field IMRT
                    if(line.Contains("Maximum␣MU"))
                    {
                        dmax = Convert.ToDouble(line.Split('=').Last());
                    }
                    if(line.Contains("Lost␣MU␣factor"))
                    {
                        fmu = Convert.ToDouble(line.Split('=').Last());
                    }
                }
                double mu_LMC = dmax * fmu;
                bool mu_same = Math.Abs((mu - mu_LMC) / mu_LMC * 100) <
                    5;//5% tolerance
                PCheck pc3 = new PCheck();
                pc3.Check = "Check␣LMC␣MU";
                pc3.Evaluation = $"LMC␣MU:␣{mu_LMC};␣Field␣MU:␣{mu}";
                pc3.Result = mu_same ? "Pass" : "Fail";
                checks.Add(pc3);
            }

            return checks;
        }
```

Compile the action pack and save it in the proper location within the file data server. Inside of the Visual Scripting UI, load any visual script that builds a report. Select Menu→Load Action Packs and load the plan Check action pack. Drag in a PlanSetup context element onto the Visual

Scripting Canvas. With the connection pane connect this context item to the Plan Checker 9000 and then to the ToTable action pack. In the ToTable Action pack, select the properties of the custom class as the table rows to enumerate the class list (Figure 7.25. Then plug the ToTable action pack into another To Report action pack.

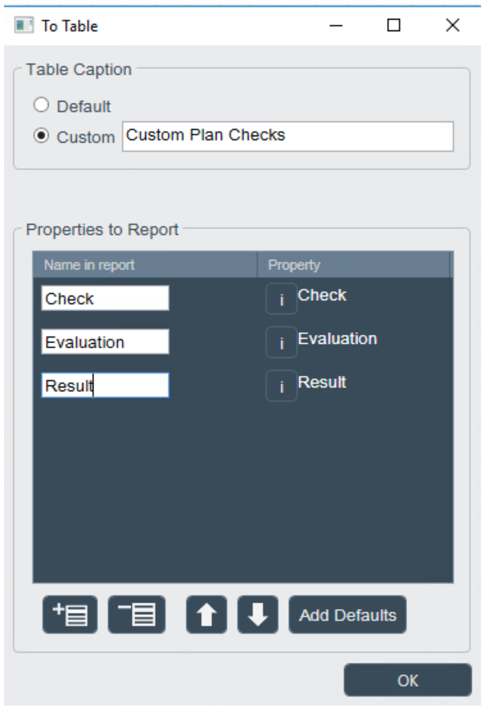

To Table

Table Caption

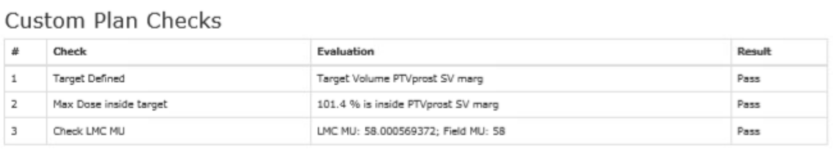
Default [✓ Custom] Custom Plan Checks

Properties to Report

Name in report	Property
Check	Check
Evaluation	Evaluation
Result	Result

Figure 7.25: Using the To Table Action Pack to enumerate a class list.

The output result should look as follows (Figure 7.26).


Custom Plan Checks

#	Check	Evaluation	Result
1	Target Defined	Target Volume PTVprost SV marg	Pass
2	Max Dose inside target	101.4 % is inside PTVprost SV marg	Pass
3	Check LMC MU	LMC MU: 58.000569372; Field MU: 58	Pass

Figure 7.26: Output from our custom action pack.

7.5 Visual Scripting Administration

This section focuses on tips on how to modify and organize your scripts and how certain features built into the Eclipse Treatment Planning System will impact your scripting environment. This is not an all inclusive list of ESAPI, Visual Scripting, or ARIA environment features that impact scripting; always have an understanding of how changing system features with administrative priveledges will impact the treatment planning and oncology information systems.

7.5.1 Approvals with Visual Scripting

As of version 15.1, there has existed a setting inside of the System and Facilities workspace within RT Administration in the System Properties Tab labelled **Approvals Required for Read-Only**

Scripts. Generally, read-only scripts are safe to run from a data integrity standpoint, but since they're dealing with sensitive and potentially patient identifying information, care should still be taken to the data security potential risk with running ESAPI scripts. Therefore this option to force approvals on read-only scripts is an optional measure by which the Eclipse Administrator can enforce extra securities. More general information on approvals for ESAPI scripts can be found in Chapter 2 on **ESAPI Basics**.

This selection will impact the running of Visual Scripts. For one, each custom action pack that is created must be approved. The *Assembly File Version* must be unique for each modification made to the script, and the script must be approved using the script approval window from the Tools menu. There are also a few built-in Visual Scripting libraries that will need approval. In version 15.5, the path to the binaries are as follows `C:/ProgramFiles(x86)/Varian/RTM/15.5/VisualScripting/ActionPacks`. In this location there are two action packs that are crucial to the running of most visual scripts, *VMS.TPS.VisualScripting.ElementLibrary.dll* and *VMS.TPS.VisualScripting.ReportActionPack.dll*. Backing up one folder level to the VisualScripting folder, *VMS.TPS.VisualScripting.ElementInterface.dll* must also be approved in order to run most Visual Scripts.

7.5.2 Action Pack Modification

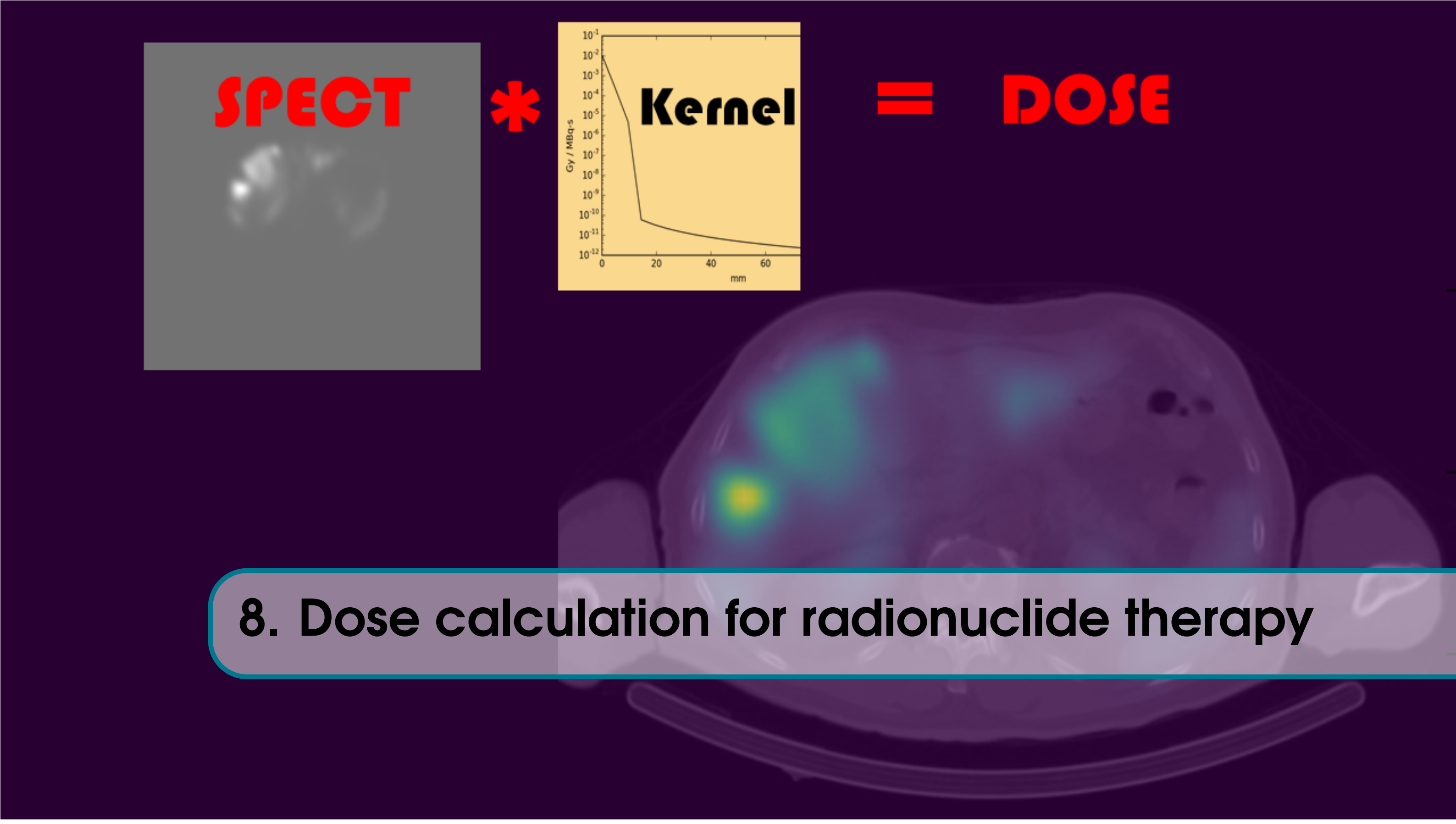
It is important to always save a version of your Visual Script without the inclusion of a custom action pack. Modifications made to the custom action pack can have consequence on the entire Visual Script. If during modification the name of the parent class of the action pack is changed, the class that starts with the name *YourActionPackElement*, or if the key-value pair relationship is modified in the *m_options* variable, then the script will be unable to open within the Visual Scripting UI. An error message will display and that script will become unmodifiable. Modifications to the execute method can be made without this worry.

When making modifications to the Execute method, Eclipse and ARIA must be completely closed at least once to allow for the removal of the action pack from Eclipse memory. After the change to the script has been made, and the script recompiled, simply replace it in the *UserDefinedActionPacks* folder described in section 7.4.3.

7.5.3 Favoriting Visual Scripts

Within the menu of the Visual Scripting Workbench, there exists an option **Add to Favorites**. When selecting this option with a Visual Script of choice open, the Visual Script will be added to the Tools menu of External Beam Planning for the user who was logged into Eclipse when the Add To Favorites option was selected. Unlike traditional ESAPI scripts, this will not allow the user to assign a hotkey and these Visual Scripts will not be seen globally by all users.

If the script is to be viewed and run by all users, it can be saved into the *Published Scripts* folder for visibility. This folder location is in the same File Data Server as the custom action packs. The Visual Script should be taken from `//<servername>/va_data$/ProgramData/Vision/VisualScripting/UserScripts/GUID_code` and saved into the following location `//<servername>/va_data$/ProgramData/Vision/PublishedScripts`. The script in question will have the user-defined saved name with an extension of vs.xml. In order to give this script a hotkey, simply open the Tools$\rightarrow$ Script menu from External Beam Planning, find the Visual Script, and add to favorites from this location.



8. Dose calculation for radionuclide therapy

Joakim Pyyry, D.Sc.

In this section we explore how to calculate dose distributions and storing results with ESAPI as an evaluation dose. With this approach we can utilize all the dose visualization and DVH tools from within the Eclipse user interface.

Radionuclide therapy is an internal radiotherapy technique which relies on certain biological mechanisms to provide a higher concentration of radionuclides in the vicinity of cancerous cells in order to produce a higher radiation dose. Often radionuclide therapy relies on radiolabeled carriers like liposomes, antibodies, or nano-particles to localize in tumors [21]. The radionuclides are administered systemically via circulation, or into cavities.

The radionuclides that are useful for therapy undergo three modes of decay: beta, alpha, and electron capture or isomeric transition by emission of Auger and Coster-Kronig electrons. Selection criteria, in addition to the mode of decay, include energy of released particles, chemical properties, production methods, as well as biological behavior. Similarly to other modes of radiotherapy — knowledge of the absorbed radiation dose is needed to assess the toxicity and efficacy of radionuclide therapy. The absorbed dose is a macroscopic concept, but for radionuclide therapy microdosimetry — study of radiation energy deposition in microscopic volumes — may be indicated for low-energy auger-emitters [10]. In this chapter we will calculate the absorbed dose with the knowledge of the spatial distribution of the activity.

8.1 Distribution of activity in radionuclide therapy

Overall treatment planning and dosimetry of radionuclide therapy requires an estimation of the radionuclide activity distribution in the patient over time [16, 19]. This cumulative activity distribution can be derived form nuclear medical imaging techniques (planar gamma camera, PET, and SPECT) and pharmacokinetic data. The image data has to be quantified so that absolute activity counts needed for cumulative activity distribution can be obtained. Additionally, pharmacokinetic modeling complements the estimation of cumulated activity in the image. The basic pharmacokinetic modeling of radiolabeled MoAb has been reviewed by Strand, Zanzonico, and Johnson [18] and a software package to calculate cumulated activities has recently been developed by Kletting et al. [11].

In this example we use a SPECT-CT image to determine the spatial distribution of radionuclide inside the patient. This information with an independent determination of the biological clearance and half-life of the radionuclide allows to determine the absorbed dose. To determine the biological clearance one needs to conduct a time-series of SPECT images or Whole-body images.

8.2 Dose calculation in radionuclide therapy

The absorbed dose in radionuclide therapy has commonly been calculated using biokinetic data from a diagnostic tracer study, using a method developed by the MIRD[20]. The traditional MIRD method includes a model patient phantom where various organ-to-organ contributions of dose have been pre-calculated with the Monte Carlo method to solve the above transport equations. When applied in a clinical setting, the MIRD method fails to include patient-specific anatomy and non-homogenous distribution of the radionuclide inside a given organ. A computer implementation of the methodology that is currently widely used is called OLINDA [17].

Absorbed dose distribution $D(\vec{r})$ at point $\vec{r}$ of known cumulative activity distribution $\tilde{A}(\vec{r})$ in a homogenous medium can be calculated with a convolution integral [9]:

$$D(\vec{r}) = \int_{V'} \tilde{A}(\vec{r}\,')k(|\vec{r}-\vec{r}\,'|)dV', \tag{8.1}$$

where $k(r)$ is a point source dose kernel (i.e. a spherically symmetric dose distribution of point source of unit cumulative activity. The kernels can be obtained from Monte Carlo simulations, analytical calculations, or physical measurements. Dose kernels have been constructed using the Monte Carlo method, for instance by Furhang, Sgouros, and Chui [8] and Reiner, Blaickner, and Rattay [14], or from cross sectional material data by Leichner [12].

The convolution integral is effectively computed discretely using the FFT as investigated by various authors, both in the realm of brachytherapy and radionuclide therapy dose calculations [5, 9]. The 3D kernel and the 3D activity map are transformed into the Fourier space, where the convolution is calculated by multiplication and transformed back to the spatial domain using an inverse FFT providing the solution to the convolution integral.

8.3 Image data manipulation

In order to prepare the dose calculation of the radionuclide therapy we need import the CT image from our SPECT/CT study that was performed after the injection of the radionuclide. After performing the import we will create an external beam treatment plan in Eclipse, this plan is used later to create the evaluation dose object that allows us to store dose values back to Eclipse from our dose calculation performed in our script. We will use PyESAPI to create Python script to calculate the dose. For more information on PyESAPI see Chapter 6.

The example case that is used here is a tracer study of 1 MBq ^{177}Lu injection. The SPECT image was acquired one hour after the injection. We start by importing the SPECT image using PyDicom library (that can be installed by `pip install pydicom`. After reading the DICOM SPECT image from file we store the pixel data into a NumPy array called spectData and plot it using matplotlib. The figure is shown in figure 8.1

Code 8.1

```
import numpy as np
import pydicom
from matplotlib import pyplot as plt
```

```
spectImage = pydicom.read_file("SPECT_Lu177_tracer.dcm")
spectData = spectImage.pixel_array
plt.imshow(spectData[40,:,:], cmap='gray')
plt.show()
```

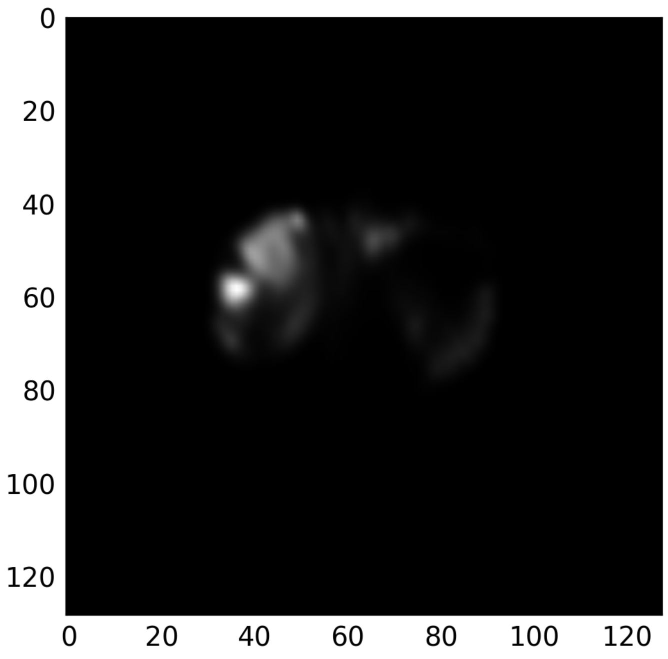


Figure 8.1: The center transversal slice of the SPECT study. The 3D SPECT image is used determine the activity distribution for the dose calculation.

8.4 Setting up the dose kernel

In this case we are looking at ^{177}Lutetium (^{177}Lu) radioisotope that has desirable physical properties. ^{177}Lu is a medium-energy β-emitter (490 keV) with a maximum energy of 0.5 MeV and a maximal tissue penetration of <2 mm. ^{177}Lu is a reactor produced radiometal that emits low-energy γ-rays at 208 and 113 keV with 10 and 6% abundance respectively. The gamma emission from ^{177}Lu allows for SPECT imaging. ^{177}Lu has a physical half-life of 6.73 days.

The dose kernel reproduces the data from Reiner, Blaickner, and Rattay [14]. We set it up in the code by setting up a look-up table every 10 mm to capture the values of the r^2 normalized dose-kernel values. After that the code interpolates the values as it constructs the 3d kernel from the look-up table and diving by the r^2 of the voxel distance. The resulting shape of the 3d-kernel from the center of the voxel grid is plotted in a logarithmic scale in figure 8.2.

Code 8.2

```
from scipy import interpolate
```

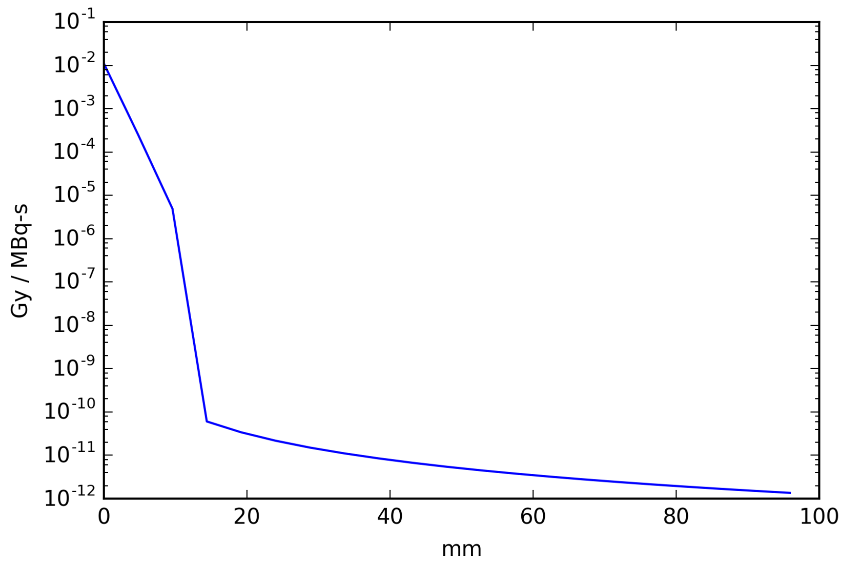


Figure 8.2: The combined radial beta and photon dose kernel for ^{177}Lu radionuclide is plotted.

```
#sub-routine to setup 3D kernel
def createKernel(size, x, y, res):
        kernel = np.zeros((size, size, size))
        center = np.array(kernel.shape) / 2.0 * res
        fkernel = interpolate.interp1d(x, y)

        for i in range(kernel.shape[0]) :
             for j in range(kernel.shape[1]) :
                     for k in range(kernel.shape[2]) :
                             pos = np.array((i,j,k))*res
                             distance = np.sqrt(sum((pos-center)**2))
                             if distance < 0.01:
                                     kernel[i,j,k] = fkernel(0.0001)
                             else:
                                     kernel[i,j,k] = fkernel(distance) /
                                          distance**2
        return kernel

#create a look-up table every 10 mm to capture shape of dose-kernel
x = np.arange(0, 601, 10, dtype='float')
y = x.copy()
y[:] = 1.25e-8  #all values
#Gy/MBqs at 10 mm
y[0] = 11.0 * 1e-3 #at center Gy/MBqs at center including the beta-dose

res = np.array((spectImage.PixelSpacing[0], spectImage.PixelSpacing[1],
     spectImage.SpacingBetweenSlices))
kernel = createKernel(42, x, y, res) #3d-kernel at SPECT image
    resolution

#plot the kernel
xx = np.arange(0,21)*res[0]
plt.plot(xx, kernel[21:,21,21])
plt.yscale('log')
plt.ylabel('Gy / MBq-s')
plt.xlabel('mm')
plt.show()
```

8.5 Dose calculation

After constructing the dose kernel we can calculate the dose by convolving the kernel with the activity distribution. We can use the fast Fourier transformation convolution method in-built from SciPy.signal (results of the dose distribution can be seen in Figure 8.3). In addition to the shape of the dose distribution determined by the dose kernel and activity distribution the actual scale of the dose values is relative to the cumulative activity at each voxel. The actual determination of the cumulated activity is intricated and requires observation of the biological clearance of the radiopharmaceutical. Here we assume infinite biological half-life and the same behavior of all the voxels (this will result in a over-estimation of the dose level).

Code 8.3 — Dose calculation.

```
from scipy import signal

total = np.sum(spectData)
#total cumulated activity based on the injected acitivity
#and half-life. In this case we assume biological half-life to be #
    infinity. Fully decayed activity is 1.44*T(1/2).
injectedActivity = 1 #1 MBq
halfLife = 6.7 * 24 * 3600 # in seconds
cumulatedActivity = injectedActivity * 1.44 * halfLife
activityScaler = cumulatedActivity / total #Cumulated activity per voxel

dose = signal.fftconvolve(spectData*activityScaler, kernel)
```

8.6 Evaluation dose

The final step is to store the dose back to Eclipse and the database via ESAPI. As mentioned earlier we imported a CT image from the SPECT/CT study and created an external beam treatment plan in Eclipse. We use the ESAPI function to open the patient with id '1000' and navigate to the newly created course 'Radionuclide' and plan 'Plan template'. We then need to instruct ESAPI to enter the editing mode where modifications can be made to the data objects. We create a new treatment plan 'PlanEvalD' and use function `CreateEvaluationDose()` for the plan to create the dose matrix where we can store our dose values.

The evaluation dose matrix is created centered to the CT image study which also happens to be the center of our SPECT study that was used as the basis of the activity map for our dose kernel convolution, thus the calculated dose distribution share the location of the central voxel with the evaluation dose matrix. However, since the resolution and the dimensions of the the dose matrices are different, we need to re-sample the calculated dose matrix to the new grid . The re-sampling is dose is performed inside the loop in the code 8.4 where we calculate the corresponding co-ordinates `x_source, y_source, z_source` of the source matrix to look-up values and store in the planes of the evaluation dose matrix. The matrix values are stored back to the Eclipse data model with the `SetVoxels(z, voxelPlane)` method where each plane of voxels is set by looping over all the planes (`z`-values).

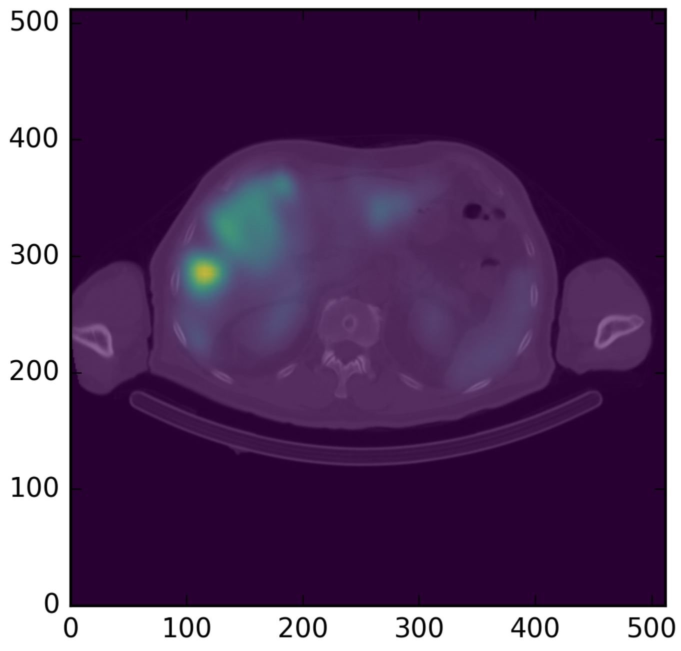


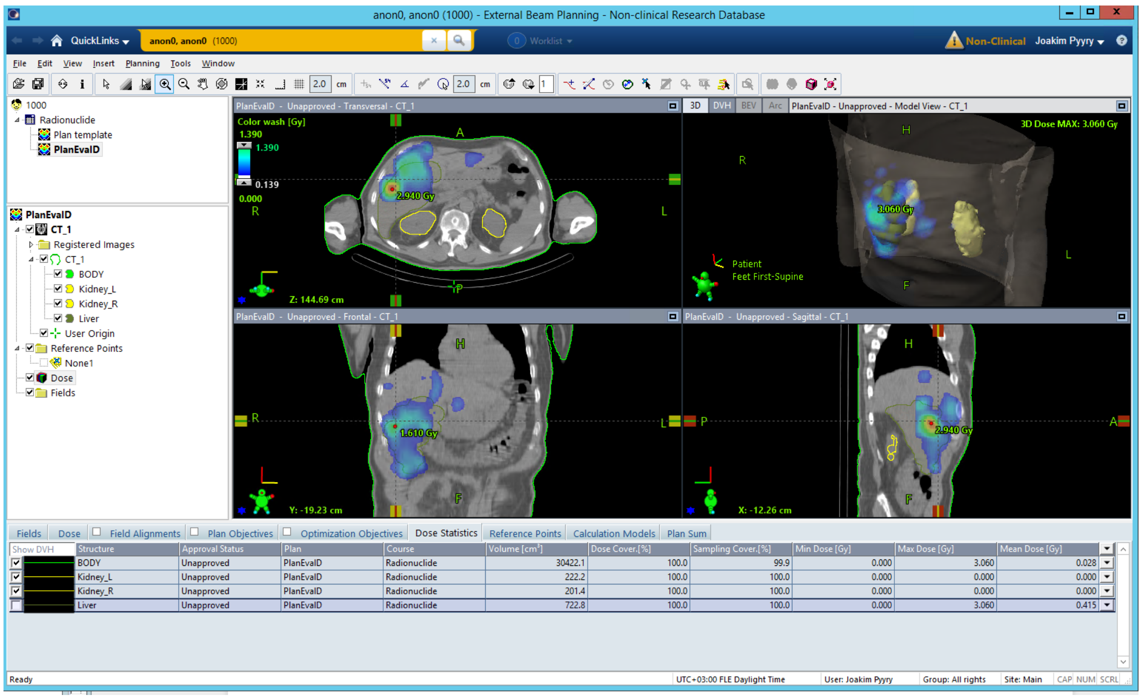
Figure 8.3: The calculated radionuclide ^{177}Lu absorbed dose distibution visualized overlaid on a CT image.

After all the operations we store the changes to the database with `app.SaveModifications()` method call. After that we can load the plan in the Eclipse user interface and see the evaluation dose visualized inside Eclipse as depicted in Figure 8.4.

Code 8.4 — Storing evaluation dose via ESAPI..

```
#open the patient, course and plan
patient = app.OpenPatientById('1000')
course = patient.CoursesLot('Radionuclide')
plan = course.PlanSetupsLot('Plan template')

#start editing patient data and then create a new plan with evaluation
    dose object
patient.BeginModifications()
plan2 = course.AddExternalPlanSetup(plan.StructureSet)
plan2.Id = 'PlanEvalD'
evalDose = plan2.CreateEvaluationDose()
#setup resolutions and sizes of the dose matrix
destSize = np.zeros(3, int)
destSize[0] = evalDose.ZSize
destSize[1] = evalDose.XSize
destSize[2] = evalDose.YSize
destRes = np.zeros(3)
destRes[0] = evalDose.ZRes
destRes[1] = evalDose.XRes
destRes[2] = evalDose.YRes
```

```
#create the .NET array to store dose values to pass to ESAPI
from System import Int32, Array
voxelPlane = Array.CreateInstance(Int32, destSize[1] , destSize[2])

#copy the dose object each dose plane at a time the dose matrices
#are of different spatial resolution but share a common center
source = dose
resS = np.array((spectImage.SpacingBetweenSlices,spectImage.PixelSpacing
    [0], spectImage.PixelSpacing[1]))
#calculate corner co-ordinates. The centers are assumed to be aligned
destStart = -0.5 * destRes * destSize #corner

for z in np.arange(destSize[0]):
    z_mm = destStart[0] + destRes[0] * z
    z_source = z_mm / resS[0] + source.shape[0] / 2.0
    for y in np.arange(destSize[2]/2):
        for x in np.arange(destSize[1]):
            x_mm = destStart[1] + destRes[2]  * x
            y_mm = destStart[2] + destRes[2]  * y
            x_source = y_mm / resS[1] + source.shape[1] / 2.0
            y_source = x_mm / resS[2] + source.shape[2] / 2.0
            voxelPlane[x, y] = int(source[np.int(z_source), np.int(
                x_source), np.int(y_source)])
    #sets voxels at each plane of the evaluation dose matrix
    evalDose.SetVoxels(z, voxelPlane)
#store back modifications to the db
app.SaveModifications()
```

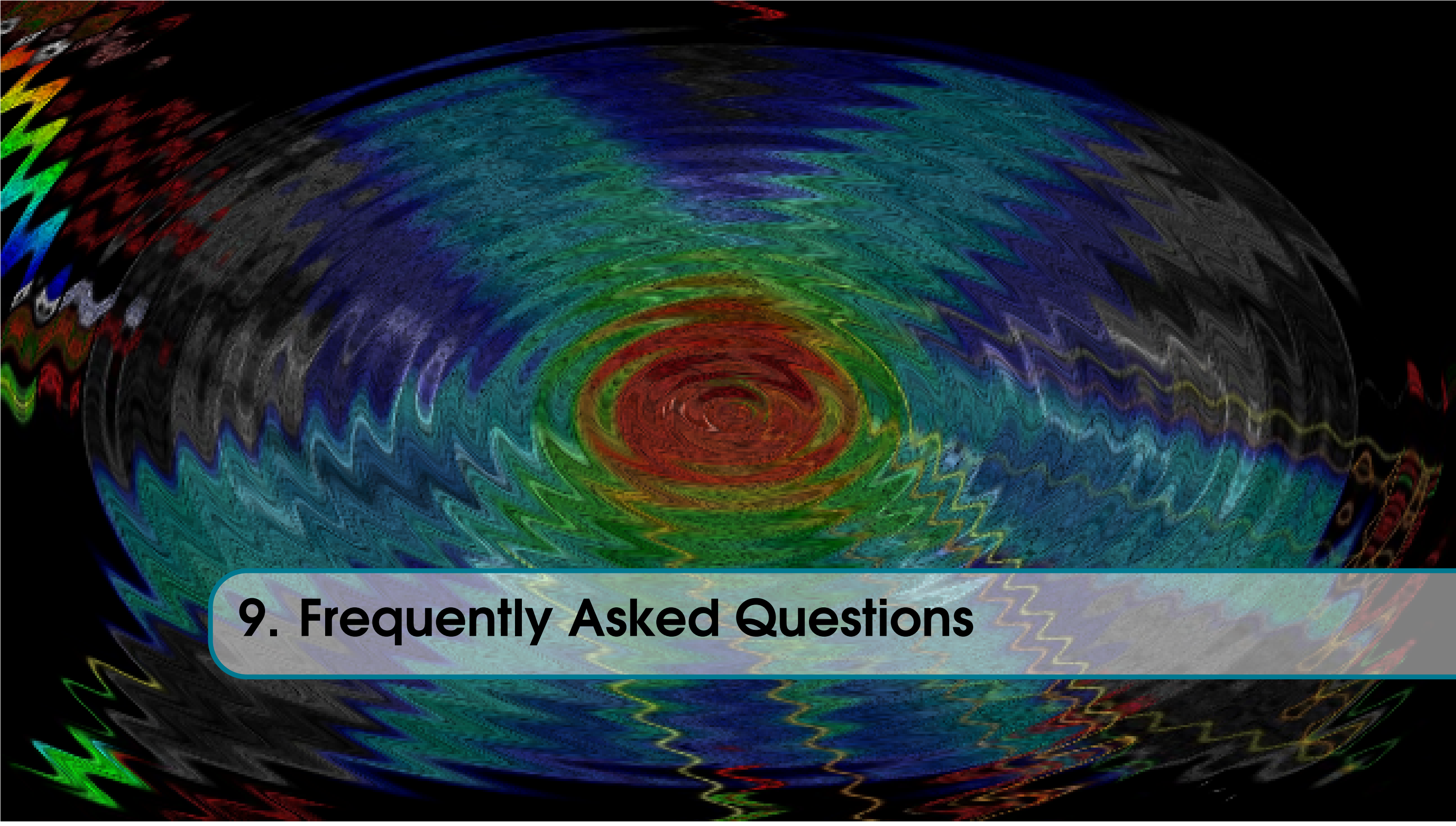


Show DVH	Structure	Approval Status	Plan	Course	Volume [cm³]	Dose Cover.[%]	Sampling Cover.[%]	Min Dose [Gy]	Max Dose [Gy]	Mean Dose [Gy]
✓	BODY	Unapproved	PlanEvalD	Radionuclide	30422.1	100.0	99.9	0.000	3.060	0.028
✓	Kidney_L	Unapproved	PlanEvalD	Radionuclide	222.2	100.0	100.0	0.000	0.000	0.000
✓	Kidney_R	Unapproved	PlanEvalD	Radionuclide	201.4	100.0	100.0	0.000	0.000	0.000
☐	Liver	Unapproved	PlanEvalD	Radionuclide	722.8	100.0	100.0	0.000	3.060	0.415

Figure 8.4: After the evaluation dose is stored, it can be visualized and analyzed within the Eclipse user interface.

II

Appendix



9. Frequently Asked Questions

WAYNE KERANEN

9.1 Eclipse Scripting API FAQ

9.1.1 General

Q: Can we script dose calculation?

A: Dose calculation cannot be scripted with the clinical version of Eclipse Scripting API in versions below 15.1.1. Eclipse 15.1.1 added the automation feature set which allows for scripting dose calculation in a clinical system.

Q: Can we use script to operate planning? Setup beams? Modify structures?

A: The Eclipse Scripting API for Research Users allows this in version 13+ on a non-clinical Eclipse research system. Eclipse Automation feature set allows this clinically in version 15.1.1 and later.

Q: Can I import fluence computed outside to Eclipse for final dose computation and MLC segmentation?

A: See methods Beam.SetOptimalFluence(...) and ExternalPlanSetup.CalculateLeafMotions(...) (15.1.1 and later).

Q: I'd like to be able to automate the creation of our verification plans. Currently we have to export the DICOM dose plane of each field. I've looked into version 11 and I can't get a single beam's dose, nor can I easily export in DICOM. How about v13.6?

A: Creation of Verification Plans is possible in a non-clinical Eclipse research system with ESAPI for Research Users v13.6 and clinically in v15.1.1 and later. Beam dose is available in the clinical ESAPI in v13.6, see method Beam.Dose(...). For access to planar dose information, see the dose profile features in clinical ESAPI (method Dose. GetDoseProfile(...)).

Q: Is there a published document for script constructs?

A: The Eclipse Scripting API Reference Guide on MyVarian explains how to get started with scripting, and the Online Help gives very detailed information about the objects, methods, and properties in the API. The Online Help is installed on Eclipse workstations and is also available through the Eclipse help menus in v13.5 and later. The online help is also provided in the Eclipse developer package. Please send an email to eclipsedeveloper@varian.com to request a copy of the Eclipse developer package.

Q: Is it possible to have scripts to import images and doses into Eclipse?
A: Images and doses cannot be imported with the Eclipse Scripting API. The DICOM DB Daemon allows for scripted import of images and dose, however, and may be an option to investigate.

Q: How do I access the script wizard in Eclipse?
A: It's available on the start menu under folder "Varian/Eclipse Scripting API" and the folder is also installed on the desktop. The script wizard comes as part of the Eclipse developer package and can be installed on any Windows computer. Note that to be able to run scripts you need Eclipse installed on the computer.

Q: Is it possible to use a script to automatically create a Dynamic Document for a patient, populated with their Eclipse plan info?
A: Yes. Use Eclipse Scripting API together with the Dynamic Documents web service. Both are available for clinical use in v11+.

Q: What advantages does Visual Studio give over something such as SharpDevelop?
A: Varian targets and tests scripting with the Visual Studio environment and does not do the same with other tools like SharpDevelop.

Q: Can we export DICOM images via the scripting interface ?
A: No, but the DICOM DB Daemon can be scripted with tools like DCMTK, EvilDICOM, or FO-DICOM to do this.

Q: Does the scripting function work for Eclipse for protons? When will the scripting API be available for protons?
A: Yes, some level of proton scripting is available when accessing only dose and DVH information for proton plans in v11. Getting access to technical proton plan information became available in the API in v13.7. Automation for protons is under development, not yet released.

Q: Is there a scripting forum for posting additional questions and problem solving?
A: In addition to OncoPeer, you can also refer to the open source community site @ www.variandeveloper.com (https://varianapis.github.io/).

Q: Does Varian write scripts for clinical users?
A: Varian Professional Services now has the capability to consult with clinical users. Please send an email to eclipsedeveloper@varian.com for more information.

Q: Can you import/export deformation vector fields?
A: Not with scripting, but you can with DICOM and DICOM DB Daemon.

Q: Can you use the ESAPI to produce images of DRRs with beams-eye-view representations of the field apertures and MLCs?
A: Yes, you can use the image profile methods available in ESAPI to ray-trace through the image volume to create DRRs and then a bit of 3D programming to layer the BEV field apertures and MLC on top of those DRRs but this is not a trivial solution.

Q: Is SmartAdapt scripting readily available?
A: Yes, SmartAdapt Scripting API is included at no additional cost with the SmartAdapt product in version 13.0+.

Q: Is there any way to attach the Visual Studio debugger to a plugin?

A: Yes, but you would do this outside of Eclipse. Create an ESAPI standalone executable project that loads your ESAPI plug-in and use the debugger on the standalone executable project. For a demonstration of this, see the open-source project PluginTester. See https://github.com/VarianAPIs/samples/blob/master/Eclips

Q: Is it possible to launch batch files or VBScripts from within Eclipse scripts?

A: Yes, see the C-sharp programming language help at msdn.microsoft.com for details on how to do this.

Q: Can I script and import Monte Carlo dose and LET values generated somewhere else, into Eclipse plan dose grid/or replace it?

A: It is possible to update the dose values of an Evaluation Dose dose grid (v13.0 - Eclipse Scripting API for Research Users, v15.1.1 available clinically).

Q: Can we use scripts to anonymize patient DICOM and export images?

A: Yes, though this is not trivial and requires use of the multiple technologies, including the DICOM DB Daemon. Export and anonymize was one of the guru track projects at Developer Workshop 2.0. The source code for this project was made available to the community as open-source software and is available here: https://github.com/VarianAPIs/samples/tree/master/webinars

Q: Is it possible to install the Eclipse API library on a PC without installing Eclipse?

A: Yes, send an email to eclipsedeveloper@varian.com with the version you need and Varian will email it to you. You will have access to all the class libraries and online Help but you would not be able to run the code on that PC as a Standalone exe. You could develop a plugin and then run it your Eclipse environment you just would not have access to the debugging tools in Visual Studio.

Q: Does Varian support the installation of Visual Studio on Eclipse thick client? Is Visual Studio approved by Varian to install on Eclipse standalone workstations?

A: Varian recommends that developers use a non-clinical Eclipse workstation for development purposes. These environments are relatively inexpensive. Call your local sales manager to purchase one. There you can install Visual Studio, do your coding, debugging, and test as needed without the risk of impacting your clinical environment.

Q: What is minimum Eclipse version to use Scripting wizard? What are the requirements for scripting? Is there a minimum ARIA version too? What material resources are needed to use scripting? (e.g. workstations, software)

A: It was first made available for version 11. All you need is a development environment (i.e. Visual Studio) and the ESAPI installer to start scripting.

Q: It was mentioned that Service Engineer has to install the API Interface? What is the Cost for the API Interface, or is it free?

A: A Varian service engineer must install the Eclipse Scripting API components on any clinical system. Developers are welcome to install these components themselves on their own non-clinical development systems. You can get the Eclipse developer package by sending an email to eclipsedeveloper@varian.com requesting it, along with the version(s) you need, and the institution you work at. The developer package is available since v13.6. Please see the Eclipse Scripting API Reference Guide, Chapter 4 for more details.

Q: Is there a way to pull dosimetric volume data from a plan sum?

A: Yes! There is a class called PlanSum which can be used similarly to a PlanSetup. PlanSum inherits many of the same methods and properties as PlanSetup. You could actually take the method we wrote in the webinar and overload it with PlanSum. All you have to do is copy and paste that method over again and change “PlanSetup plan” to “PlanSum plan” and its ready to be reused for a PlanSum.

Q: As a total beginner, where is the best place to start learning C# and scripting for Eclipse?

A: Just jump in! You could start by trying to follow along with the presentation and write out that code yourself. Then just play around with it and explore what’s there using the Autos and Watch functions from the “Debug”-> “Windows” tab. Explore the API and reference guide for things you are interested in. For C#, there are lots of text books and tutorials out there too but there is really no substitute for the real thing.

Q: How is it best to manage ESAPI upgrades/new releases?

A: ESAPI reference guides walk you through important changes to the new release but they are typically backwards compatible. It’s also important to get a testing environment up and running before you go live so that you can make sure your scripts work in the new version. Certainly it depends on how reliant your department is on the script functionality.

Q: Is the API available only is C#? Is there C++ API? Python?

A: Only C# at this time in clinical systems. Python support was introduced in 15.5 for research mode systems. See pyESAPI chapter in this book.

Q: Where do you typically store clinically used scripts?

A: We have a production folder on our Fullscale server where tested, approved scripts are housed. Version 15.1.1 will also give you the ability to sign and approve your code so that you can ensure only clinically approved versions are used in practice.

Q: What other software can we use besides visual studio to write C# code?

A: You can use any text editor or integrated development environment you prefer. My experience is only in Visual Studio so that’s all I can personally recommend.

Q: Can you export data/results/output to a pdf file or word document?

A: Yes, there are other resource files you will need. I use an API called “PDF Sharp” which has great methods for generating and printing PDFs. For Microsoft word you need to open up the reference manager and in the assemblies tab there should be an option you can choose for “Microsoft.Office.Interop.Word” if you have selected the “.NET Programmability Support feature” on install of Office. If not you may have to search for the files in the ProgramFiles folder on your C drive.

Q: Can you access the data from off-line review and export them for analysis?

A: Offline review is part of Aria so it is not a function of the Eclipse Scripting API. The API may be expanded in the future to encompass Aria and offline review.

Q: Do you need an active eclipse license on the machine to run the scripts?

A: You can develop with just the API and IDE installed, but to test or run the scripts you need at least a non-calculation installation of Eclipse.

Q: How to best manage code you want to reuse in more than one script? How to have methods in a separate file?

A: You can add new classes by going to “Project” in the task bar, then clicking “Add Class”. Add your new code there, then you can instantiate that class (ex. NewClass c = new NewClass();) back inside the Main or Execute methods of which ever project you are in. You can then access its member methods and properties by writing “c.nameOfMethod”.

Q: I noticed that you have ARIA 15.1, and visual scripting. How do you evaluate the use of it, or you prefer doing the code by writing rather than GUI?

A: We started scripting before visual scripting was available. Visual scripting is an excellent tool for getting started without any actual coding experience, however editing the base C# code gives the coder lots of flexibility so we are continuing down that path.

Q: Can you nest several scripts? (e.g. make a script that could call another script depending on some choices)

A: We have not tried that but it may be possible. Our practice is to reuse portions of code by copying classes into new projects and using the logic operators to handle different scenarios.

9.1.2 Q & A for Webinar - Eclipse Scripting: Intro to Automation & Visual Scripting

Presenter, Wayne Keranen, responds to questions asked during this webinar held on April 6, 2018.

Q: Tell us more about Varian Marketplace™. It seems Varian Marketplace is only available in few places. When this is going to change? When will it be available in Canada?

A: Varian Marketplace™ is a digital marketplace for Varian software product extensions to solve real-world clinical and operational problems. The marketplace lets developers share their Varian-related applications and other content with Varian customers, and offers Varian customers access to solutions created by their peers. No comment on when it will become available in other markets.

Q: What is meant by non-clinical Eclipse™? Is a training box considered a non-clinical Eclipse system? If so, will have this enable me to get the automation license on v15.5?

A: Yes, having an Eclipse training system present in your enterprise should enable you to get the Eclipse automation license in 15.5. The training system is a non-clinical Eclipse workstation that can be used for script development. See Eclipse Scripting API Reference Guide, Chapter 9, topic “Configuring a Non-Clinical Development System”.

Q: Is Eclipse Automation available only in research licenses?

A: “Eclipse Automation” refers to the feature set that includes:

- Clinically Writeable Scripting
- Script Approval.

Eclipse Automation is available for clinical use since Eclipse 15.1.1.

Q: What do you mean by approving scripts?

A: Eclipse has a new feature since 15.1.1 where the QA Authority in your clinic must approve write-enabled scripts before they can be used on a clinical system. This feature is found under the External Beam workspace; “Tools/Script Approval” menu. See the Eclipse Scripting API reference guide (P1021698-003-C), Chapter 9, found on MyVarian for more details. Script Approval is optionally available for read-only scripts and can be configured in the RT Administration workspace.

To configure an Eclipse system so that read-only Eclipse scripts also require approval by the QA Authority in your clinic, (assuming you have the proper user rights):

1. Open RT Administration using a system administrator account.
2. Click System and Facilities.
3. Click System Properties.
4. Select the Approval Required for Read-only Scripts check box.
5. Select File > Save All to save changes.

Q: Do you have Visual Studio installed in the Eclipse box?

A: Visual Studio and other development tools should be installed only on your development system; which is typically a non-clinical Eclipse workstation.

Q: Is approval required to test scripts on the non-clinical Eclipse?

A: No. When an Eclipse system is configured as a non-clinical system script approval is not required. See Eclipse Scripting API Reference Guide, Chapter 9, topic "Configuring a Non-Clinical Development System".

Q: Is there a script that has been developed that will evaluate the targets and OAR objectives from the RX in ARIA to assess the quality of the plan from the DVHs in the finalized plan in Eclipse?

A: No, but there is an open-source script available that evaluates DVH metrics using templates. Radformation ClearCheck is a commercially available Eclipse Script that can be found on Varian MarketPlace. ClearCheck evaluates DVH metrics for targets & OARs.

Q: Does ESAPI now have the capability of writing?

A: Write-enabled Eclipse Scripting is available for clinical use since Eclipse 15.1.1.

Q: Can you develop using ESAPI without Eclipse?

A: You can create scripts using on a system that does not have Eclipse installed, but you need Eclipse installed on the system to debug and run scripts. See Chapter 4 in the Eclipse Scripting API reference guide (P1021698-003-C). You can send an email to eclipsedeveloper@varian.com to get a copy of the Eclipse Scripting API installer.

Q: Where can I find the reference guide with details on ESAPI automation features and methods you mentioned (e.g. FitToStructure, AddMLC)?

A: The Eclipse Scripting API Online Help (P1021731-003-C) provides detailed information on the properties and methods available in the Eclipse Scripting API. The online help is installed with the Eclipse Scripting API installer. . See Chapter 4 in the Eclipse Scripting API reference guide (P1021698-003-C). You can send an email to eclipsedeveloper@varian.com to get a copy of the Eclipse Scripting API installer.

Q: What is the difference between the raw and final proton scanning spots functionality in scripting?

A: Raw spot list is the proton scanning spots after optimization but before dose calculation and post-processing. Final spot list is the post-processed scanning spot that is ready for treatment delivery.

Q: Do graphic scripts need approval?

A: The QA Authority in your clinic must approve write-enabled scripts before they can be used on a clinical system. This feature is found under the External Beam workspace; "Tools/Script Approval" menu. See the Eclipse Scripting API reference guide (P1021698-003-C), Chapter 9, found on MyVarian for more details. Script Approval is optionally available for read-only scripts and can be configured in the RT Administration workspace.

Q: When a script creates a new structure and tries to add it to the structure set but a structure with that name already exists, does it overwrite or cause an error?

A: An error is thrown by Eclipse in this situation.

Q: Must you have a standalone non-clinical box to access the script wizard?

A: The recommended script development system is a non-clinical Eclipse workstation. You can send an email to eclipsedeveloper@varian.com to get a copy of the Eclipse Scripting API installer.

Q: Is it possible to have some user interaction built into a script or external executable (e.g. to expand the PTV and subtract from the rectum, the user selects the size of the expansion from a dropdown)?

A: Yes, Eclipse Scripting API uses fully functional C#.NET as its language so you can program scripts that have sophisticated user interfaces.

Q: Are there benefits for clinical use using saved template planes?

A: (I presume this question means are there benefits of using scripts vs templates) Scripts can give you more detailed control over plan creation vs template plans, and could potentially automate some manual work in the process.

Q: Can scripts be created for custom MLC sequences or modifying existing MLC sequences?

A: Yes. You can autofit MLCs, you can run Leaf Motion Calculator, you can also set leaf positions with method BeamParameters.SetAllLeafPositions(...).

Q: Are there best practices for managing scripting that you can share, version control, repositories, testing controls? What is a suggested clinical organization?

A: We highlight an example process in the Eclipse Scripting API Reference Guide (P1021698-003-C), Chapter 9. Varian recommends that you follow good engineering and clinical development practices when developing scripts that are intended for clinical use. Best practices are shared in the EC301 class where a clinical script developer participates in the last part of the class to share practice and experience.

Q: What are the differences and limitations in operating scripts/stand-alone apps with Citrix versions of Eclipse?

A: See next answer.

Q: How do I develop a script if we are on Citrix? What do I need to take into account when scripting in a Citrix environment?

A: The best way to develop a script for Citrix is to acquire an Eclipse non-clinical workstation and develop on that. Citrix is one approach to provide remote windowing in a Windows environment. The main differences between a Citrix-hosted Eclipse and a locally running Eclipse are the way the system resources are accessed. The file system exposed by Citrix will look different since it is the file system of the Citrix server that Eclipse is running on, and not the local computer used to interact with the Citrix-hosted Eclipse. Local IT administrators may have certain system features locked down that are not usually protected on local systems, like the ability to dynamically load DLLs, for example.

Q: If we want to run compiled scripts in our Citrix environment, do we need to also develop (compile) scripts in the same Citrix environment (e.g. install Visual Studio Community in Citrix)?

A: You can compile scripts on any non-clinical Eclipse system compatible with your Citrix-hosted Eclipse. You can also compile scripts on any Windows computer that has Eclipse Scripting DLLs which are compatible with the Eclipse environment hosted on Citrix. You can send an email to eclipsedeveloper@varian.com to get a copy of the Eclipse Scripting API installer.

Q: Is there anything in the works to make running standalone executables in the Citrix environment easier to launch standalone executables more and more people moving to the Citrix environment?

A: Some members of the community have created script runners that make launching scripts in Citrix easier. See also the ESAPI method ScriptEnvironment.ExecuteScript.

Q: Will the tables and documents work in a Citrix environment?

A: Yes, Visual Scripting works in a Citrix environment.

Q: Will scripts need revisions as Eclipse changes versions?

A: We introduced a strict versioning scheme for ESAPI 15.5, so scripts developed for 15.5 and later will not need to be revised or even recompiled for subsequent versions.

Q: How would you recommend a beginner using version 11 get started?

A: We recommend you start your scripting journey by attending the EC301 - Eclipse Scripting API in Varian's Las Vegas training center. Another less intense option is to join us in Nashville for Developer Workshop 2018.

Q: Can I use scripts written for v15 in v13.7?

A: No

Q: Can you explain the differences and what is available in 15.0 vs 15.1 vs 15.5? Is visual scripting available only on 15.5, or earlier versions of 15.1?

A: Visual Scripting was introduced in 15.0 and is included in all subsequent versions. Eclipse Automation was introduced in 15.1.1 and is included in all subsequent versions. Our latest cleared and shipping Eclipse release is 15.5, which is why the webinar highlighted 15.5. 15.0 – Visual Scripting Introduced. 15.1.1 – Eclipse Automation cleared for clinical use. 15.5 – Eclipse Automation extended to include more features like MLC autofitting.

Q: If I am using 13.7, what is the best way to debug a plugin or binary script?

A: Use a standalone executable script that loads your plugin script so that you can debug the standalone executable script with Visual Studio. There are different examples of how to do this out in the community. Carlos Anderson details one on his blog (http://www.carlosjanderson.com/run-and-test-plug-in-scripts-from-visual-studio/).

Q: How difficult to transfer my ESAPI codes from V13.5 to V15.5?

A: Not difficult. API changes have been mostly additions, and incompatibilities are noted in the Eclipse Scripting API Reference Guide.

Q: Can we implement a "traffic light" system in DVH evaluation like in Oncentra® or use DVH-constraints for plan eval in a visible way?

A: Yes, and we have an open source script to help you get started.

Q: Is the Visual Scripting workbench available for v13 users?

A: No. Visual Scripting is available in Eclipse 15.0 and later.

Q: What is the source code for the: Optimization structure creation & the plan creation scripts? Do these scripts work in v15.1?

A: The optimization structure creation script will work for 15.1.1. The plan creation script will not work exactly as written since we added MLC fitting in 15.5. The scripts are open source examples that can be found here: https://github.com/VarianAPIs/samples , "webinars & workshops" folder.

Q: Is ARIA® Access that you're exposing the same or different from ARIA Link?

A: Different. Aria Link was an SQL stored procedure based interface, and Aria Access is a web service. Aria Access has functionality that replaces a large part of but not all of the Aria LINK. You can find more information about Aria Access by searching MyVarian.

Q: Is using Entity Framework to access the ARIA database expected to create overheads that are unacceptable?

A: This is an advanced developer topic. Varian doesn't recommend ways to perform direct Aria DB access.

Q: Is there a way to access ARIA-related info (i.e. patient booking/scheduling) using Eclipse scripting (either as script or stand-alone)? Only way we managed was through the SQL database

A: Not so far.

Q: What is the future of AURA given the reporting features available in visual scripting?

A: AURA and Visual Scripting provide different reporting features so we expect the two will remain independently for some time. We encourage a clever programmer to create new Visual Scripting Action Packs that bring AURA information into Visual Scripting to make the two work seamlessly together.

9.1.3 Licensing

Q: Is scripting available to all users who have Eclipse or do you have to buy a license?

A: Eclipse Scripting API is available for all customers who have purchased Eclipse 11+ at no additional cost. There is a separate license file required which is included by default. If you find Eclipse scripting is not available on your v11 or later Eclipse system, please contact your local Varian support personnel.

9.1.4 Citrix

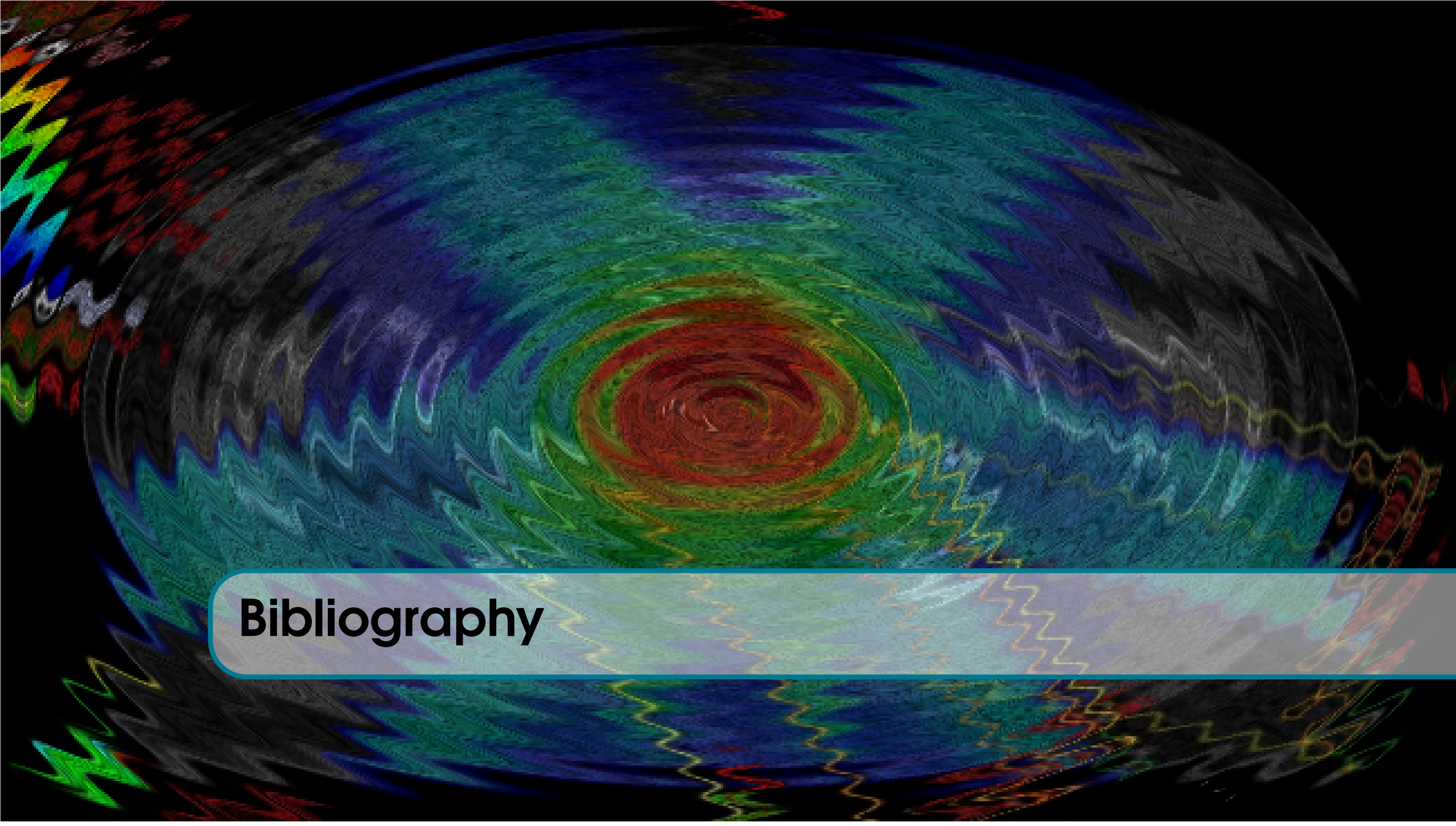
Q: We have version 11 via Citrix. How do we access the Eclipse Scripting Wizard?

A: Ask your IT support personnel to publish Eclipse Script Wizard on the Citrix system. In version 13.6 MR 0.5 and later we make the Eclipse Scripting API components available separately so that developers can install them on their own workstation. Please send an email to eclipsedeveloper@varian.com to request a copy of the Eclipse developer package.

Q Can scripting be done if using Eclipse hosted on the cloud? My site uses Eclipse 11 over Citrix. Who can I talk/email with about how scripting is different in that environment? A Yes, but there are challenges that you will need to work with your IT on, like exposing Visual Studio, the command prompt, and the Windows Explorer. The best way to develop is on a thick Eclipse client (non-clinical of course).

Q: Does scripting work when Eclipse runs over Citrix^{TM}?

A: Yes, though additional configuration is required in Citrix to allow scripts to run. To develop Eclipse scripts within a Citrix environment, you may have to ask your IT support personnel to configure Citrix. By default, Citrix environments do not let you load DLLs and run executables from foreign directories and extra configuration is required. See the link below. http://support.citrix.com/article/CTX105611 We recommend that Citrix users who will do intensive ESAPI script development acquire a non-clinical Eclipse workstation designated for development and testing. Programmers install Visual Studio^{TM} and then develop and test their ESAPI scripts on the non-clinical Eclipse workstation and eventually deploy to the Citrix clinical environment when convinced the scripts they have created are working safely and effectively. Once the scripts are thoroughly tested, they should be compiled as binary scripts (either plug-ins or standalone executables). Compiling as binary versions allows you to better control the scripts and apply versioning information so you can track your requirements and testing and any bugs found against that version number.



Bibliography

Books

[1] ISO 12052 NEMA PS3. *Digital Imaging and Communications in Medicine (DICOM) Standard*. Rosslyn, VA, USA: National Electrical Manufacturers Association. URL: `http://medical.nema.org/` (cited on page 10).

[2] Varian. *ARIA Link Reference Guide" (100060538-01)*. Palo Alto, CA, USA: Varian Medical Systems (cited on page 10).

[3] Varian. *Eclipse Scripting API Reference Guide" (P1021698-003-C)*. Palo Alto, CA, USA: Varian Medical Systems (cited on page 17).

[4] Varian. *MLC File Format Description Reference Guide" (1106064-06)*. Palo Alto, CA, USA: Varian Medical Systems (cited on page 10).

Articles

[5] A .L. Boyer and E. C. Mok. "Brachytherapy seed dose distribution calculation employing the fast Fourier transform". In: *Med Phys* 13 (1986), pages 525–529 (cited on page 122).

[6] et. al. Covington Elizabeth L. "Improving treatment plan evaluation with automation." In: *JACMP* 13 (1986), pages 525–529 (cited on page 14).

[7] Joseph O. Deasy, Angel I. Blanco, and Vanessa H. Clark. "CERR: A computational environment for radiotherapy research". In: *Medical Physics* 30.5 (2003), pages 979–985. DOI: `10.1118/1.1568978`. URL: `https://aapm.onlinelibrary.wiley.com/doi/abs/10.1118/1.1568978` (cited on page 9).

[8] E. E. Furhang, G. Sgouros, and C. S. Chui. "Radionuclide photon dose kernels for internal emitter dosimetry". In: *Med Phys* 23 (1996), pages 759–764 (cited on page 122).

[9] H. B. Giap et al. "Validation of a dose-point kernel convolution technique for internal dosimetry". In: *Phys Med Biol* 40 (1995), pages 365–381 (cited on page 122).

[10] J. L. Humm et al. “Microdosimetric concepts in radioimmunotherapy”. In: *Med Phys* 20.2 Pt 2 (Mar. 1993), pages 535–541. URL: http://view.ncbi.nlm.nih.gov/pubmed/8492762 (cited on page 121).

[11] P. Kletting et al. “Molecular radiotherapy: the NUKFIT software for calculating the time-integrated activity coefficient.” eng. In: *Med Phys* 40.10 (Oct. 2013), page 102504. DOI: 10.1118/1.4820367. URL: http://dx.doi.org/10.1118/1.4820367 (cited on page 122).

[12] PK Leichner. “A unified approach to photon and beta particle dosimetry”. In: *J Nucl Med* 35 (1994), pages 1721–1729 (cited on page 122).

[13] J Perl et al. “TOPAS: an innovative proton Monte Carlo platform for research and clinical applications.” In: *Medical physics* 39 (11 Nov. 2012), pages 6818–6837. ISSN: 0094-2405. DOI: 10.1118/1.4758060 (cited on page 9).

[14] Dora Reiner, Matthias Blaickner, and Frank Rattay. “Discrete beta dose kernel matrices for nuclides applied in targeted radionuclide therapy (TRT) calculated with MCNP5”. In: *Med Phys* 36.11 (Nov. 2009), pages 4890–4896. URL: http://view.ncbi.nlm.nih.gov/pubmed/19994497 (cited on pages 122, 123).

[15] D W Rogers et al. “BEAM: a Monte Carlo code to simulate radiotherapy treatment units.” In: *Medical physics* 22 (5 May 1995), pages 503–524. ISSN: 0094-2405. DOI: 10.1118/1.597552 (cited on page 9).

[16] G. Sgouros et al. “Treatment planning for internal radionuclide therapy: three-dimensional dosimetry for nonuniformly distributed radionuclides.” eng. In: *J Nucl Med* 31.11 (Nov. 1990), pages 1884–1891 (cited on page 121).

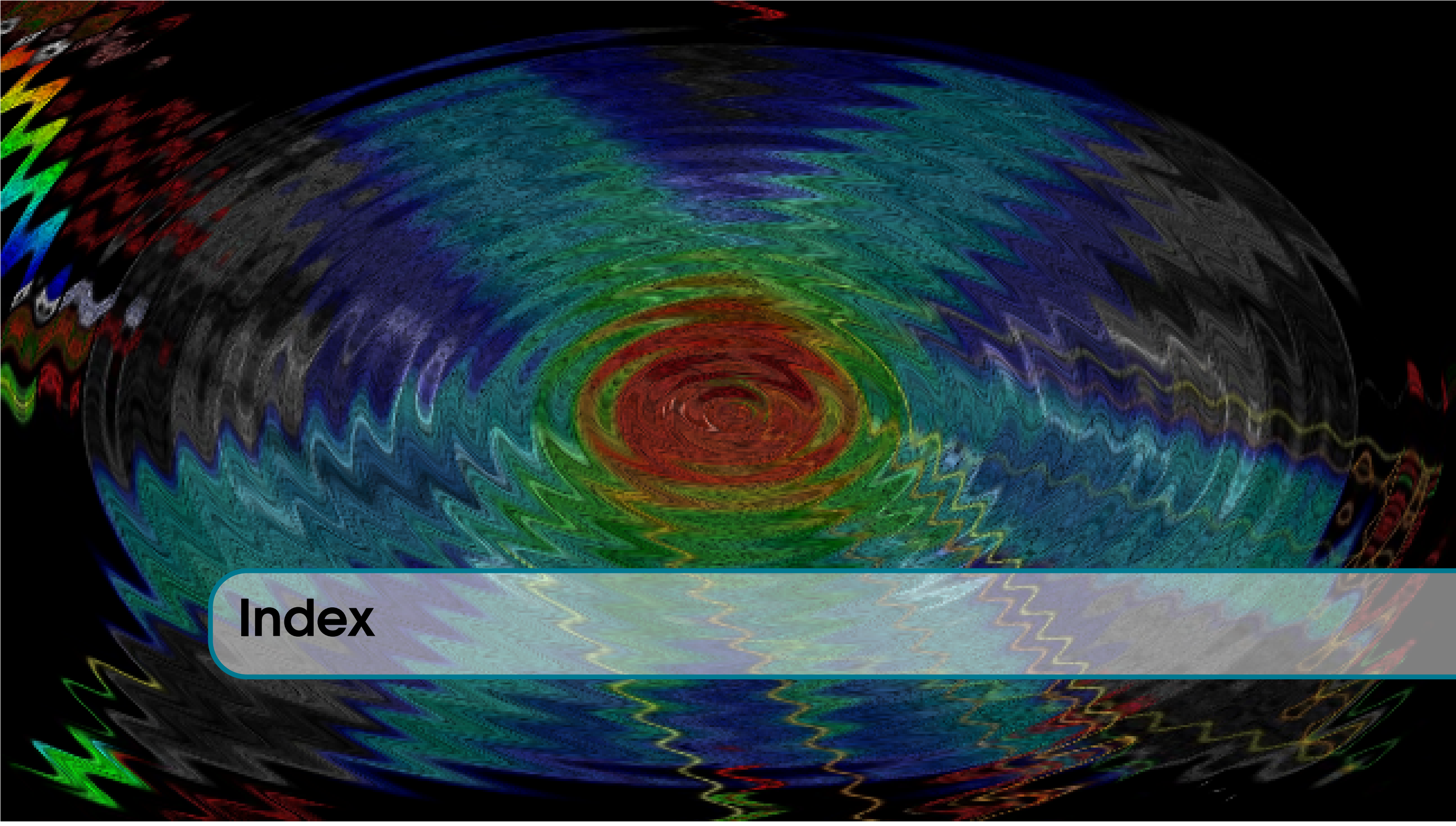
[17] Michael G. Stabin, Richard B. Sparks, and Eric Crowe. “OLINDA/EXM: the second-generation personal computer software for internal dose assessment in nuclear medicine.” eng. In: *J Nucl Med* 46.6 (June 2005), pages 1023–1027 (cited on page 122).

[18] S E Strand, P. Zanzonico, and T. K. Johnson. “Pharmacokinetic modeling”. In: *Med Phys* 20.2 Pt 2 (Mar. 1993), pages 515–527. URL: http://view.ncbi.nlm.nih.gov/pubmed/8492760 (cited on page 121).

[19] S. E. Strand et al. “Radioimmunotherapy dosimetry–a review.” eng. In: *Acta Oncol* 32.7-8 (1993), pages 807–817 (cited on page 121).

[20] E E Watson, M G Stabin, and J A Siegel. “MIRD formulation”. In: *Med Phys* 20.2 Pt 2 (Mar. 1993), pages 511–514. URL: http://view.ncbi.nlm.nih.gov/pubmed/8492759 (cited on page 122).

[21] Lawrence E. Williams, Gerald L. DeNardo, and Ruby F. Meredith. “Targeted radionuclide therapy.” eng. In: *Med Phys* 35.7 (July 2008), pages 3062–3068 (cited on page 121).



Index