This is the SolTrace help menu from SolTrace 2012 compiled as a PDF. This was created using HelpNDoc. We will continue to develop the documentation using the HelpNDoc software, and changes noted herein should be made in the original HelpNDoc file.

This file contains annotations indicating areas that need attention as part of the documentation update.

SolTrace

Copyright © 2010 National Renewable Energy Laboratory

Table of contents

Introduction 3
Migrating to 2010 4
Methodology 6
Basic Use 9
Defining the Sun 11
Geometry 12
Optical Properties 21
Tracing 23
Visualization 25
Exporting Data 27
SolScript Automation 29
- Data Variables 31
- Flow Control 34
- Arrays of Data 39
- Function Calls 41
- Input, Output, and System Access 44
- Function Libraries 46
  - Types and Data 46
  - Input and Output 47
  - Math 50
  - String Manipulation 51
  - Matrices 52
  - SolTrace Library 53

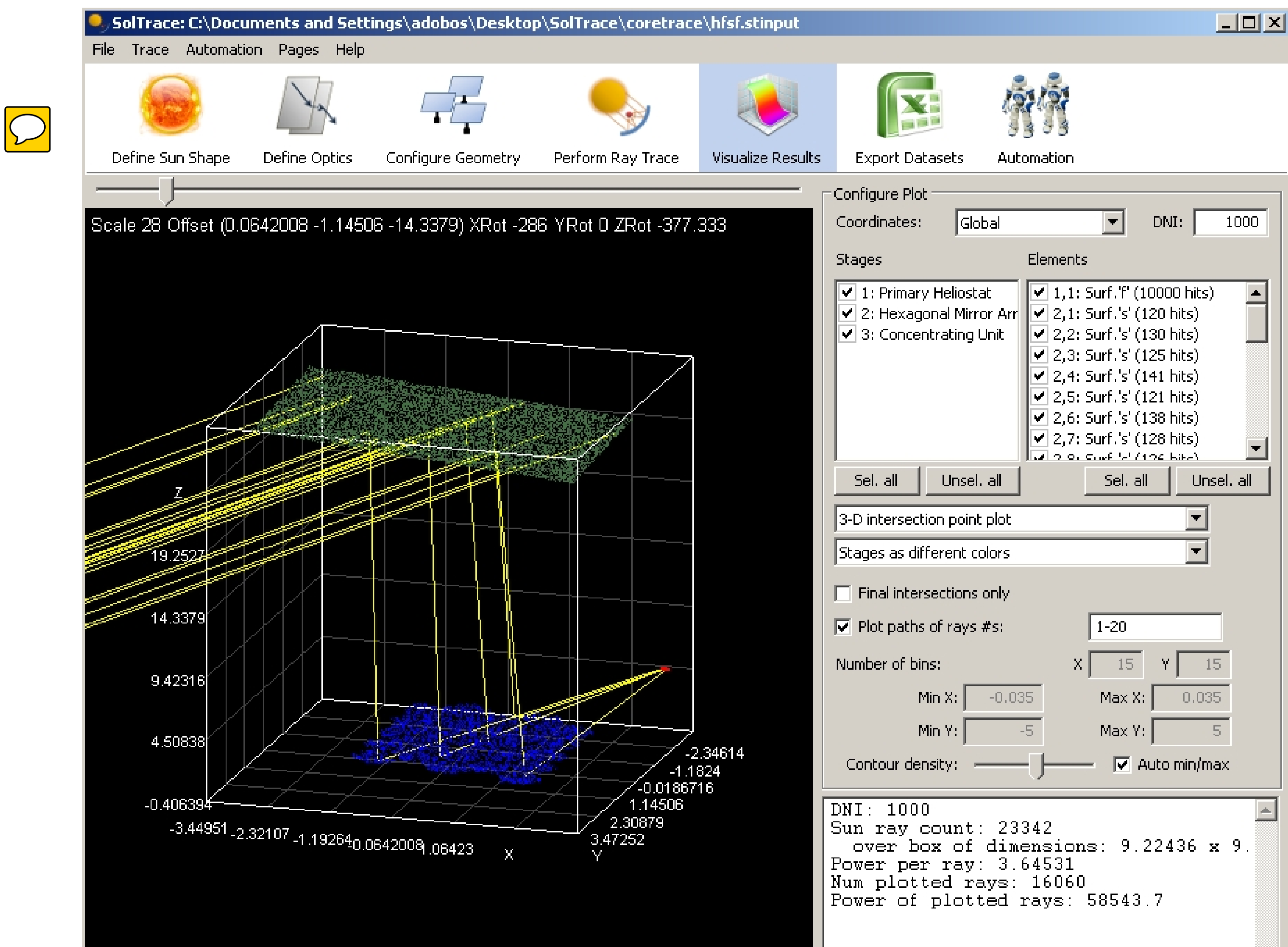
References 56

Introduction

SolTrace is a software tool developed at the National Renewable Energy Laboratory (NREL) to model solar power optical systems and analyze their performance. Although originally intended for solar optical applications, the code can also be used to model and characterize general optical systems. The creation of the code evolved out of a need to model more complex optical systems than could be modeled with existing tools such as OPTDSH (Steele et al., 1991) and CIRCE (Ratzel and Boughton, 1987).

The code utilizes a ray-tracing methodology (Spencer and Murty, 1962). The user selects a given number of rays to be traced. Each ray is traced through the system while encountering various optical interactions. Some of these interactions are probabilistic in nature (e.g. selection of sun angle from sun angular intensity distribution) while others are deterministic (e.g. calculation of ray intersection with an analytically described surface and resultant redirection.) Such a code has the advantage over codes based on convolution of moments in that it replicates real photon interactions and therefore can provide accurate results for complex systems that cannot be modeled otherwise. The disadvantage is longer processing time. Accuracy increases with the number of rays traced and larger ray numbers means more processing time. Also, complex geometries translate into longer run times. However, the required number of rays is also a function of the desired result. For example, fewer rays (and therefore less time) are needed to determine relative changes in optical efficiency for different sun angles on a given solar concentrator than say are needed to accurately assess the flux distribution on the receiver of that same concentrator. Thus the responsibility is on the user to use the code wisely and efficiently.

The application is written in C++ and uses the cross-platform wxWidgets toolkit, allowing it to run on Windows, Linux/Unix, and Mac OS X operating systems. The core tracing procedures are separated from the graphical user interface, and allow for the incorporation of the calculation code into other systems and applications. SolTrace scales exceptionally well to run on computers with multiple processors, as each individual ray can be computed independently of the others. Computers with 8 processors will generally experience close to an 8x speedup over a single processor, but this trend can be limited by various factors including the specific nature of a particular geometry or other computer hardware limitations (memory, etc).



Created with the Personal Edition of HelpNDoc: Easy CHM and documentation editor

Migrating to 2010

Overview

In 2010, SolTrace was completely recoded from the ground up to improve performance, utilize parallel processing techniques, update the user interface, and provide the framework for future interoperability with other modeling tools. This upgrade greatly expands the capability of the model to analyze large optical systems and perform complex batch simulations using the built-in scripting language. The new version is written in C++ using the wxWidgets graphical user interface toolkit, and can run on Windows and Mac OS X systems. The previous version, having been written in the Delphi programming environment could only run on Windows and experienced some compatibility issues even across versions of Windows. Screenshots of both versions are included below for comparison.

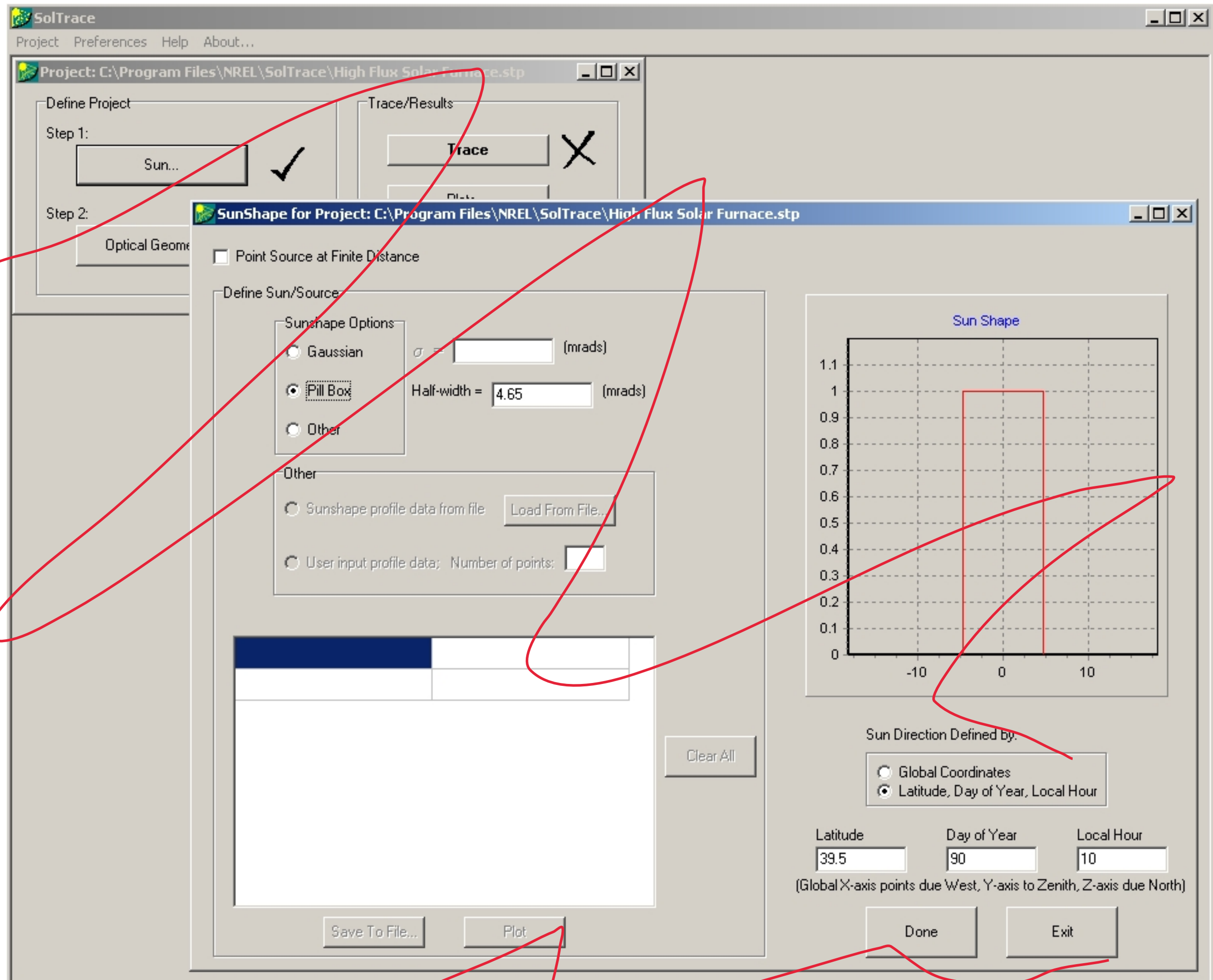


Figure. Previous versions of SolTrace (Delphi)

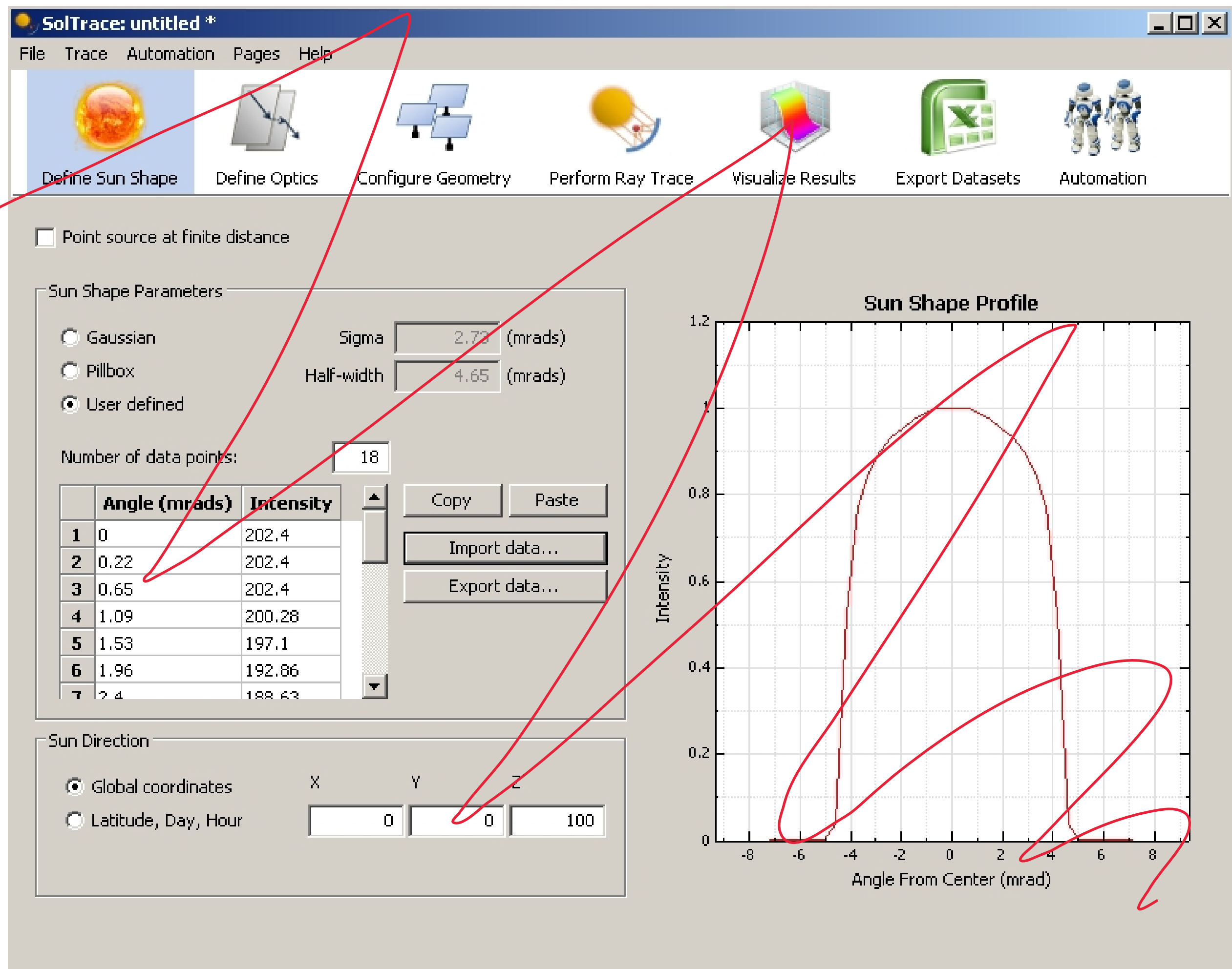


Figure. New version of SolTrace (C++/wxWidgets)

Importing Old Projects

Importing old projects in the (*.stp) file format is not possible at this time. Rather, a user must manually import the individual *.sun, *.opt, and *.geo files that comprise a complete system geometry. We provide a sample script called importgeo.ss (click on the Automation tab and open it by double-clicking it in the file browser on the left side) that will read a multi-stage .geo file and import the stage and element data, overwriting existing system geometry. It is expected that this script will work correctly in most cases, but may not fully import all stage properties (Virtual, trace through, multi-hit etc...)

Note on Visualization

Currently, no 3-D flux charts are available, and the contour plotting is somewhat more limited than in the original Delphi version. We are actively investigating ways to improve visualization in the new version of SolTrace.

Created with the Personal Edition of HelpNDoc: Full-featured Documentation generator

Methodology

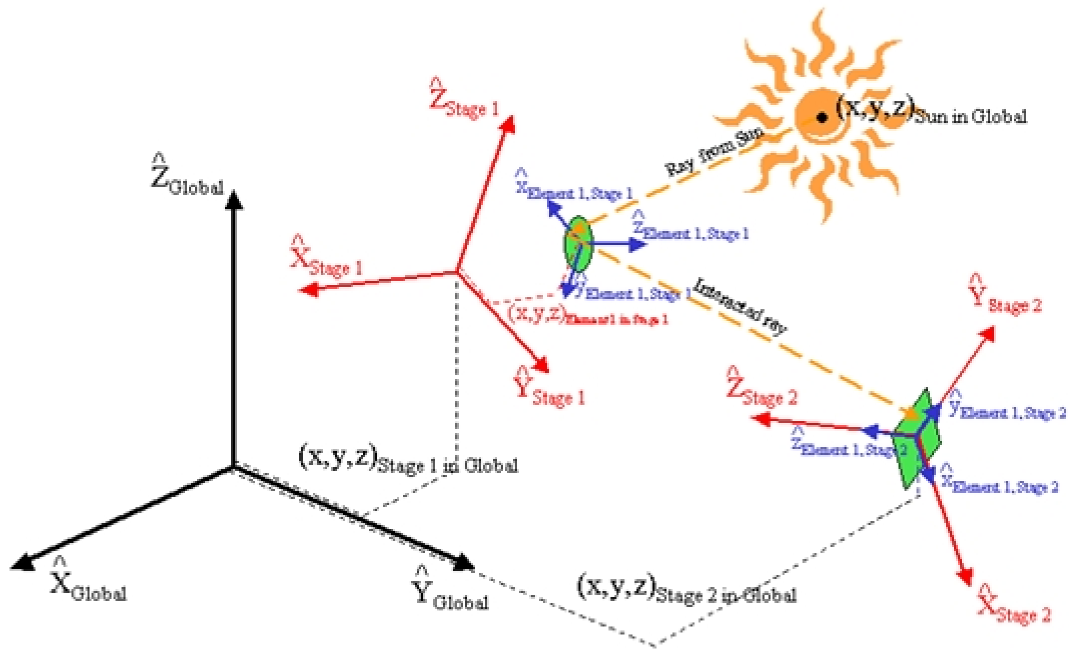
In SolTrace, an optical system is organized into “stages” within a global coordinate system. A stage is loosely defined as a section of the optical geometry which, once a ray

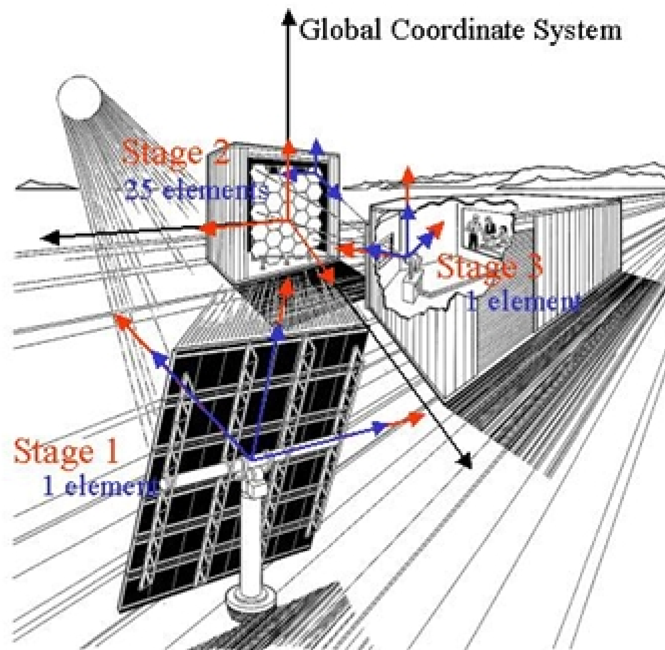
exits the stage, will not be re-entered by the ray on the remainder of it's path through the system. A complete system geometry may consist of one or more stages. It is incumbent on the user to define the stage geometry accordingly. The motivation behind the stage concept is to employ efficient tracing and therefore save processing time and allow for a modular representation of a system. The other significant benefit of stages is that they can also be saved and employed in other system geometries without the need for recalculating element positions and orientations.

A stage is comprised of "elements". Each element consists of a surface, an optical interaction type, an aperture shape and, if appropriate, a set of optical properties. The location and orientation of stages are defined within the global coordinate system whereas the location and orientation of elements are specified within the coordinate system of the particular stage in which they are defined. Stages can be one of two types: optical or virtual. An optical stage is defined as one that physically interacts with the rays. Conversely, a virtual stage is defined as one that does not physically interact with the rays. The virtual stage is useful for determining ray locations and directions at various positions along the optical path without physically affecting ray trajectory. Elements defined within a virtual stage therefore have no optical properties because they do not interact with the rays. Optical stages consist of elements which interact with the rays potentially altering their trajectories. These elements have optical properties and interaction types associated with them. Beyond this, optical and virtual stages are identical in how they are defined and used. Stages can be duplicated and moved around as groups of elements and then saved for use in other system geometries.

SolTrace uses three right-handed coordinate systems: the global coordinate system, the stage coordinate system and the element coordinate system. These are illustrated in Figure 1a. Each element in a stage has a local coordinate system (i.e. location and orientation) defined relative to the stage coordinate system. Each stage has a coordinate system defined relative to the global coordinate system. As shall be described later, the direction of the sun is defined relative to the global coordinate system. Currently, the sun direction is input in either vector form, or in time-of-day, day-of-year format with latitude specified. Light rays are generated from the sun and then traced sequentially through each stage in the geometry. The position and direction of each ray in each stage is stored in memory for later processing and output.

The NREL High Flux Solar Furnace is shown as an example of a multi-stage, multi-element system in Figure 1b. Note the global (black), stage (red) and element (blue) coordinate systems. In this example, there are a total of three stages. The first stage is comprised of one flat rectangular reflective element (the heliostat). In this case, the element coordinate system lies directly on top of the stage coordinate system. The second stage is the primary concentrator consisting of twenty-five hexagonal reflective elements each having spherical curvature. One of the twenty-five individual element coordinate systems is shown to the upper right of the stage coordinate system. The third and final stage is the sample stage (a rectangular flat target) in the HFSF control room. Again, the element coordinate system occupies the same location and orientation as the stage coordinate system in this case. The details of element/stage definition are presented later.





Figures 1a-b. SolTrace Coordinate Systems

The stage and element coordinate systems are translated from the global coordinate system and stage coordinate system origins respectively and then oriented via three Euler angle rotations. These rotations are shown in Figure 2.

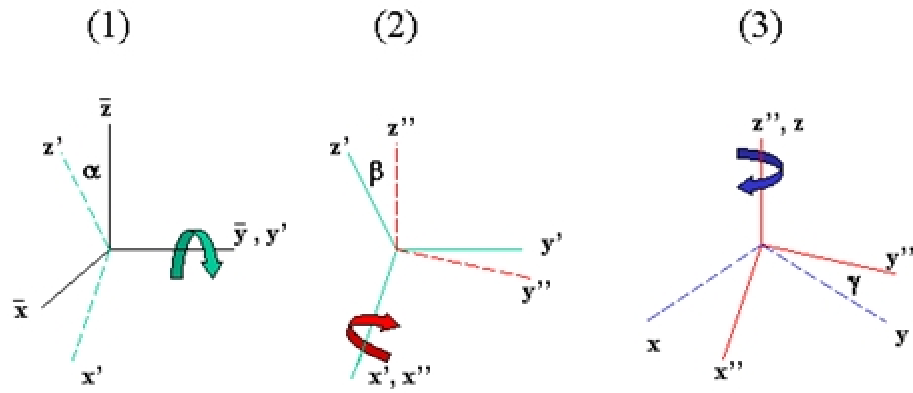


Figure 2. Generation of (x, y, z) system from the (/x, /y, /z) system after translation of origin.

The first rotation is by angle α about the y-axis, the second rotation by angle β about the x-axis and the last rotation by angle γ about the z-axis. After translation of the child coordinate system origin from the parent coordinate system, these three rotations completely specify the orientation of the child coordinate system within the parent. The first two rotations are automatically determined by specifying a point in the parent coordinate system toward which the z-axis of the child coordinate system is aligned. The last rotation of the child coordinate system about the z-axis is then specified.

Created with the Personal Edition of HelpNDoc: Free Qt Help documentation generator

Basic Use

Upon opening the SolTrace application, the main window opens with an empty project, as shown below.

Figure. Empty project upon opening SolTrace application

A SolTrace project file contains a complete set of inputs that define a model or analysis. This includes information about the sun shape and position, a set of surface optical interaction properties, and all the elements for each stage in the system. The data is stored in a plain text formatted file with the "*.stinput" extension. The results of a ray trace are not stored in this file, nor are externally defined surface types such those described by VSHOT data (discussed later). As a result, the .stinput files are generally quite small.

The icon toolbar across the top of the main window allow the user to move between various data input and output pages, shown below.

Figure. SolTrace Navigation Toolbar

The first three pages (Sun Shape, Optics, Geometry) are used to define the system. The Perform Ray Trace page is the place to configure the number of rays desired, the number of computer processors to use, and other simulation related parameters.

After a trace has completed successfully, the Visualize and Export pages are populated with the calculated ray intersection data. Several graphing options are included, as well as the ability to save the raw intersection data in comma-separated value (CSV) format,

or to export directly to Microsoft Excel on Windows computers.

The Automation page exposes advanced scripting features that allow a user to programmatically control SolTrace to configure the geometry of complex systems with hundreds of elements (i.e. a heliostat field for a power tower), or to post-process the calculated intersection points. ~~Some example scripts are provided, include one that can import old format *.geo files from previous versions (pre-2009) of SolTrace~~.

Created with the Personal Edition of HelpNDoc: Free Kindle producer

Defining the Sun

Two characteristics completely define the "sun" as the light source: the angular intensity distribution of light across the sun's disk (referred to as the sunshape) and the sun's position. The area in the upper left of the window shown in Figure 5 is the first step in defining the sunshape. Three options are available. The first two are commonly used probabilistic distributions. Although the sunshape varies widely with terrestrial location, sky conditions and time and is neither truly Gaussian nor pillbox in nature, this approximation is adequate for a large class of problems. The parameter defining a Gaussian distribution for the sun's disk is the half-angle σ. The parameter for the pillbox, being a flat distribution, is simply the half-angle width.

The third option allows the use to define the sunshape profile as a series of intensity datum points (Neumann et al., 2002). Since the sunshape is axisymmetric, only half of the profile (from the sun's center to the edge) is required. This can be manually entered into the table seen in the lower left corner of the window. This table becomes available when the user selects "User defined" option. The number of datum points in the half profile is entered in the box to the right. The profile is entered as a set of datum points: the angular position from the sun's center in milliradians (starting with 0) and the corresponding relative intensity at that position. The intensity values can be in any units because the code scales the distribution to a peak of unity. Once entered into the table, the profile can be stored in a file for future use and later retrieved using the Import and Export buttons. The sun shape plot automatically updates as data is entered into the shape table.

Once the sunshape has been defined, all that remains is to determine the sun direction. One option is to define a point in the global coordinate system such that a vector from this point to the global coordinate system origin defines the sun direction. The other option is to define a particular site latitude and time (day of year and local hour.) From this information, the sun direction is determined assuming the z-axis of the global coordinate system points due north and the y-axis points towards zenith. This is important to remember when defining the optical geometry.

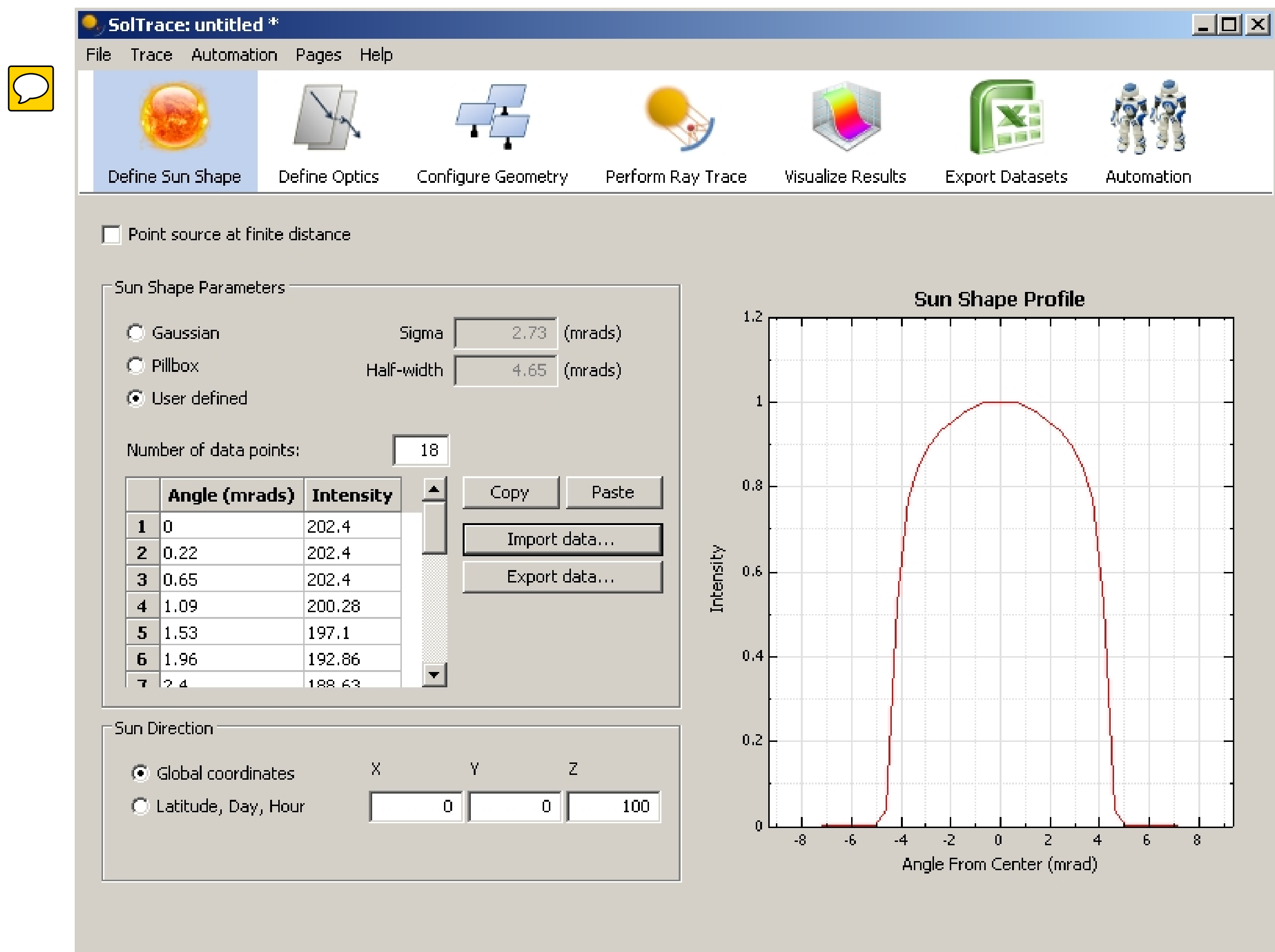


Figure 5. Defining the Sun Shape and Position

Created with the Personal Edition of HelpNDoc: Easily create EPub books

Geometry

Defining System Geometry

Once the sun is defined, the user can proceed to the next step – defining the optical geometry to analyze. The system geometry is defined on the "Configure Geometry" page, shown below. The

This section requires substantial effort. We should first introduce the geometry features, including the process of adding, removing, or inserting stages, then the process of adding geometry. The HFSF example given below can follow the general discussion. This section should be reorganized to more logically follow the workflow process in SolTrace. A suggested outline could be:
* Defining system geometry
  - Adding, inserting, and removing stages
    * Stage properties (discuss virtual stage, multiple hits per ray, and trace through options)
    * Stage global coordinates, including Edit Z rotation dialog
    * Data transfer options
  - Element editing
    * Insert, append, delete, delete all
    * Z rotation dialog
    * Aperture options (good documentation below)
    * Surface options (ditto)
    * Optics (ditto)
  - Example with HFSF

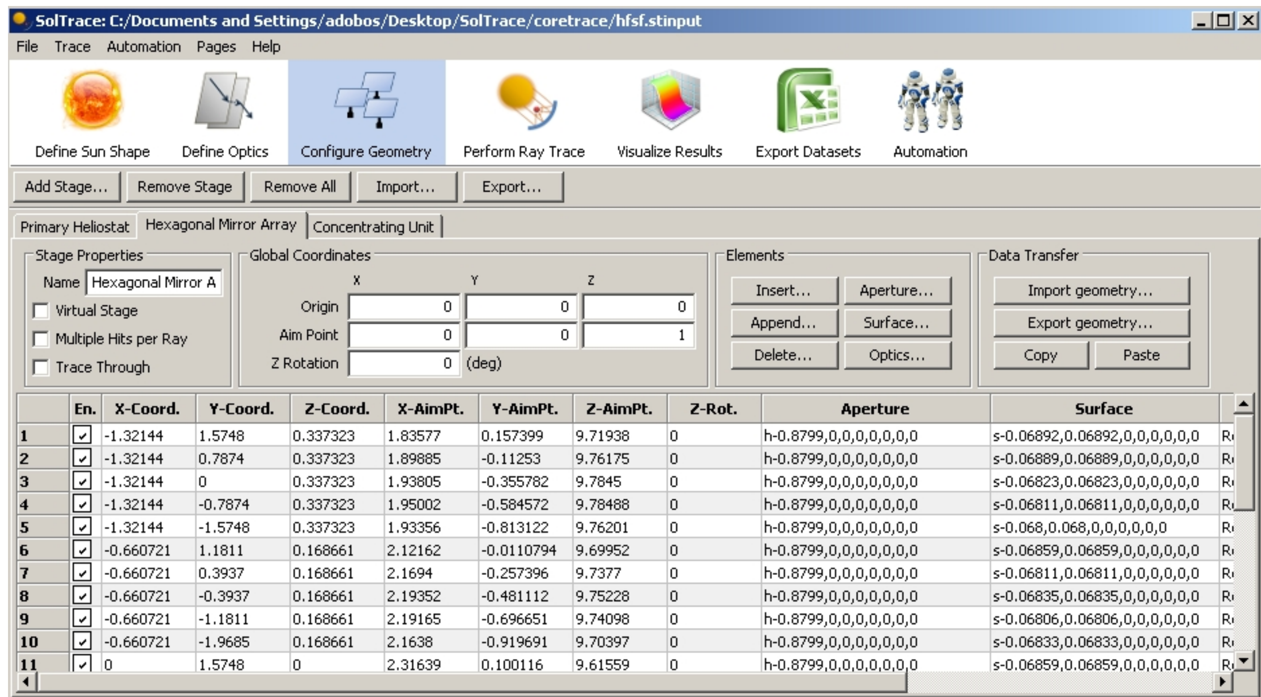


Figure 6. Optical geometry definition input page.

The input page is shown with data already entered for the sake of example. The form can now be used to design every aspect of the optical problem to be analyzed. First, stages can be added and removed using the buttons at the top of the page. Using the Import and Export buttons, specific stages can be saved and recalled later as needed. Each stage is displayed as a tab, much like worksheets in a spreadsheet. For each stage, there is a set of stage properties, position and orientation information in the global coordinate system, as well as a table of optical elements.

Calculation Procedure for Multiple Stages

Three stages have been defined in this example which is actually that of the NREL High Flux Solar Furnace shown in Figure 1b. The first two stages are of the optical type (i.e. contain actual optical elements that interact with light). Stage 1 consists of the primary heliostat which redirects sunlight to the concentrator (Stage 2.) The last stage is a virtual stage consisting simply of a flat target for the resultant flux distribution generated at the focus of the concentrator in Stage 2. Light rays will ultimately be traced from the sun to Stage 1 then to Stage 2 and finally to Stage 3. This assumes that once rays leave a stage they cannot physically re-enter that stage again by any combination of the stages following (a reiteration of the stage definition.) The user must carefully design the optical system such that the stages meet this criteria otherwise incorrect results will be obtained.

Before proceeding to the checkout buttons at the bottom of the window, mention should be made of the checkbox labeled 'Trace Through' which appears (already checked by default) at the top of each optical stage worksheet (with the exception of Stage 1 if it is an optical stage.) Simply put, if this box is checked it means that rays continue to be traced through the rest of the system even if they miss all the elements of this particular stage. If not checked, then rays which miss this stage completely are no longer traced through the rest of the system. More information on the tracing procedure is available

in the 'Tracing' section.

Stage Position and Orientation

The first set of three inputs is the location (in x,y,z) of the stage coordinate system origin within the global coordinate system. The set of four numbers below this determines the orientation of the stage coordinate system within the global coordinate system. The first three numbers of this set define a special point within the global coordinate system. A vector from the stage origin to this point defines the z-axis of the stage coordinate system. The last degree of freedom to be defined is then the rotation of the stage coordinate system about this z-axis. This is entered as the fourth number in degrees (Z Rotation). Every stage must have these parameters defined within the global coordinate system.

Element Definition

Elements can be added and removed from the stage by using the appropriate Insert, Append, and Delete buttons located above the main element data table. Each element is defined on one row of the table, is identified by its row number and by the stage in which it exists.

It is possible that the user may want to “turn on” or “turn off” certain elements within a stage while doing different traces for comparison purposes. Rather than deleting all the data for a particular element and then having to re-enter it again later, the "En." checkbox can be used to enable and disable elements. If checked (the default), the element is included as part of the geometry, if unchecked the element is ignored.

The next three columns are the x,y,z coordinates of the element coordinate system origin within the stage coordinate system. The next three columns define the x,y,z coordinates of a special point within the stage called the aim point. A vector from the element origin to this aim point defines the z-axis of the element coordinate system. The ninth column is the rotation of the element coordinate system about it’s z-axis in degrees. This set of seven numbers describing the location and orientation of the element within the stage is exactly analogous to the set described above for the stage within the global coordinate system.

Apertures

The aperture column contains the description of the projected shape of the element opening in a plane perpendicular to the element z-axis. An example of a circular aperture is shown in Figure 8.

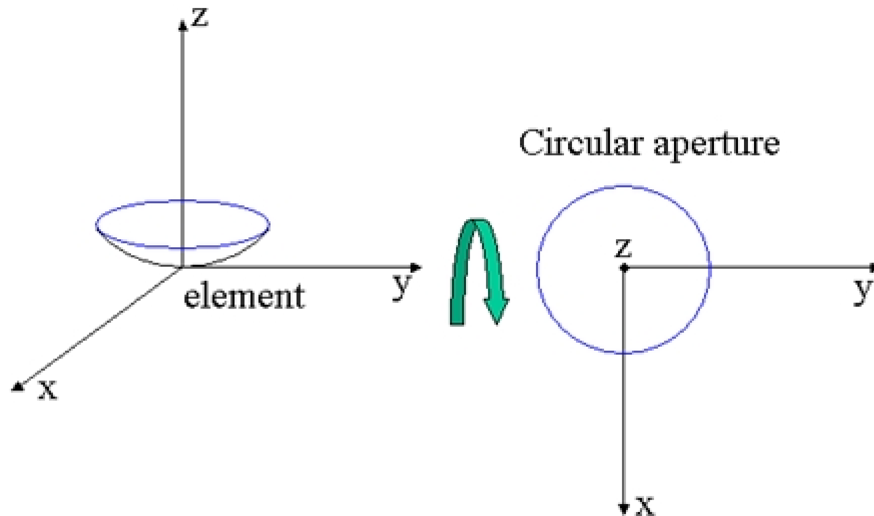


Figure 8. Aperture description

A variety of aperture shapes is available. The column entry is a text string that encodes the aperture description. The code format for the different apertures is shown in Table 1. The code can be entered manually as a string of text. A code begins with a lower case letter denoting the type of aperture shape, followed by a hyphen, and then a list of eight numbers separated by commas. Some aperture shapes require all eight parameters, while others may only require one. Shape parameters that are unused can be set to 0 and are ignored. The program prevents the user from entering an incorrect string code. An option to manually entering the code is to click on the "Aperture" button above the table. When clicked, a dialog pops up to help a user set the aperture for the currently selected element in the table, shown below.

Aperture Type	***Code***
Circular	c-# (# = diameter of circular aperture)
Hexagonal	h-# (# = diameter of the circle which circumscribes a hexagonal aperture)
Triangular	t-# (# = diameter of the circle which circumscribes an equilateral triangle)
Rectangular	r-$\#_1,\#_2$ ($\#_1,\#_2$ = width, height of rectangle)
Annular	a-$\#_1,\#_2,\#_3$ ($\#_1,\#_2,\#_3$ = inner radius, outer radius, included angle in degrees; $\#_1 < \#_2$, $0 < \#_3 \le 360$)
Single Axis Curvature Section	l-$\#_1,\#_2,\#_3$ ($\#_1,\#_2,\#_3$ = distance to inner edge in x dir, distance to outer edge in x dir, length of section in y dir; $\#_1 < \#_2$)

Table 1. Some aperture types and corresponding codes

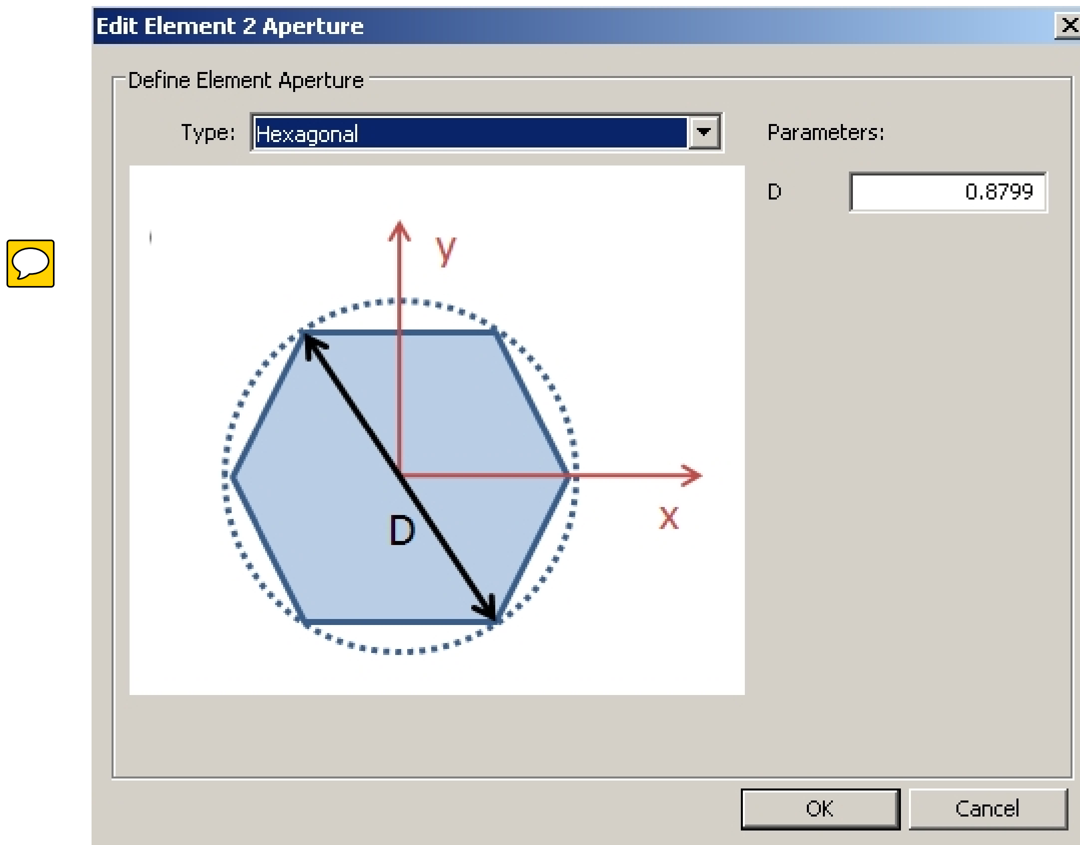


Figure 9. Aperture definition window

The drop-down list at the top of this sub-window contains a list of all the available aperture options. Selecting one displays a graphical description of the aperture and entry boxes for entering the relevant parameters for that aperture. Clicking on “OK” automatically enters the correct string code for the selected aperture into the aperture type column.

Surfaces

Each element's surface is defined by a text code in the "Surface" column of the table, similar to how the aperture is defined. A variety of surface options are available. Table 2 lists the surface options and respective text codes.

Surface Type	***Code***
Parabolic	p-#1,#2 (#1,#2 = 1/radii of curvature in x , y directions of a parabolic surface)
Spherical	s-#1,#2 (#1,#2 = 1/radii of curvature in x , y directions of a spherical surface)
Other (hyperboloid, ellipsoids)	o-#1,#2,#3 (#1,#2 = 1/radii of curvature in x, y directions, #3 = κ parameter for other surfaces)
Flat	f
Conical	c-#1 (#1 = half angle of conical surface)
Zernike Series	*.mon (surface described by Zernike series equation with coefficients in file “*.mon”)

VSHOT Data Set	*.sht (surface described by VSHOT data file "*.sht")
Finite Element Data Set	*.fed (surface described by finite element data file "*.fed")
General Spencer & Murty Equation	g-#1,#2,#3,#4,#5,#6,#7,#8 (#1,#2= 1/radii of curvature in x , y directions, #3= κ , #4-8 = $\alpha_{1\text{-}5}$)
Cylinder	t-#1 (#1 = 1 / radius of curvature; use in conjunction with aperture code "l-0,0,#2" where #2 is length of cylinder)
Polynomial Series (rotationally symmetric)	*.ply (surface described by coefficients of polynomial equation in file "*.ply")
Cubic Spline Interpolation (rotationally symmetric)	*.csi (surface described by discrete data points and 1st derivative boundary conditions in file "*.csi")

Table 2. Surface types and corresponding codes

The "Zernike Series" surface type is described by

$$Z(X, Y) = \sum_{i=0}^{k} \sum_{j=0}^{i} B_{ij} X^j Y^{i-j}, \quad k = \text{order of monomial}$$

The Zernike series file format is (where N is the order and the B's are the coefficients):

N
$B_{0,0}$
$B_{1,0}$
$B_{1,1}$
$B_{2,0}$
$B_{2,1}$
$B_{2,2}$
.
.
.
$B_{N,N}$

There are a total of (N+1)(N+2)/2 coefficients for the Zernike Series.

The "General Spencer & Murty" formulation is

$$F = Z - c\rho^2/[1 + (1 - \kappa c^2\rho^2)^{1/2}] - \sum_{i=1}^{N} \alpha_i \rho^{2i} = 0,$$

where N need not exceed 5 in most cases (and in fact has been hardwired to this number.)

If terms α_i are omitted from this equation above, it reduces to the "Other" equation representing revolved conic sections. For surfaces described by "Other" or "General Spencer & Murty", the above general equation reduces to

$F = Z - c\,(\rho^2 + \kappa Z^2)/2 = 0$, where c = 1/radius of curvature at vertex, $\rho^2 = X^2 + Y^2$

κ determines the surface type according to the following:

For $\kappa < 0$ surface is a hyperboloid

For $\kappa = 0$, surface is a paraboloid

For $0 < \kappa < 1$, surface is a hemi-ellipsoid of revolution about major axis

For $\kappa = 1$, surface is a hemisphere

For $\kappa > 1$, surface is a hemi-ellipsoid of revolution about minor axis

Surfaces specified by "Spherical" , "Parabolic" or "Conical" are subsets of the more general "Other" surface type designation ~~which in turn is a subset of the even more general "General Spencer & Murty" formulation.~~

The Polynomial Series option is described by the equation

$$Z(\rho) = \sum_{i=0}^{N} C_i \rho^i \quad \text{, where} \quad \rho = (X^2 + Y^2)^{1/2}$$

A surface described by this choice is rotationally symmetric. The polynomial series file format is (where N is the order and C's are the coefficients):

N
C_0
C_1
C_2
.
.
.
C_N

A surface described by Cubic Spline Interpolation of a data point set (ρ_i , Z_i) is also rotationally symmetric. The data required are the data points themselves and the derivatives (or slopes) at the first and last points (boundary conditions). The cubic spline interpolation file format is (where N is the number of data points):

N
ρ_1 Z_1
ρ_2 Z_2
.
.
.
ρ_N Z_N
$dZ/d\rho$ at point 1 $dZ/d\rho$ at point N

Special considerations for specifying the aperture and surface types exist and are listed below:

1) It was originally intended that spherical and parabolic surfaces could have different curvature in the x and y directions. Unfortunately, this was never implemented, and the framework for specifying these was never removed. Thus, for ‘s’ and ‘p’ surfaces, the 2nd parameter must be equal to 1st parameter or equal to zero (see item 2 below).
2) The only aperture type allowed for single axis curvature surfaces specified by ‘p-#,0’ or ‘s-#,0’ is the ‘l’ type aperture. Other aperture types, such as rectangular (‘r’ type), will not work with single axis curvature surfaces.
3) Except for VSHOT, Zernike monomial ~~and FEA surface~~ descriptions (all of which can be asymmetric), only rotationally symmetric surfaces, planes, trough sections and cylinders can be described by the above analytical formulations.
4) The cylindrical surface (type ‘t’) requires the ‘l’ aperture to be specified with the first two parameters set equal to 0 and the third being the length of the cylinder: ‘l-0,0,length’.
5) ~~The FEA surface description has not been fully implemented and so is not described.~~
6) The VSHOT file format is not described as this type of file is only available through NREL testing.
7) If tracing rays into a closed surface such as a cylinder it is recommended that a “virtual” window be placed at the end (or entrance) to the cylinder. This can be accomplished with a flat circular element of optic type = 1 (i.e. refractive surface) where the refraction indices for both the front and back side of the element are both = 1 (i.e. zero refraction),the transmissivity = 1.0 (i.e. perfect transmission) and the optical errors are set to very small values (e.g. <0.001 mrad). This surface does not physically interact with the rays (i.e. does not affect their path), but corrects for numerical problems associated with ray origins being far from the entrance to the cylinder. It also provides a convenient surface to flux map the entrance and compare with the exit.

As in the case of the aperture type description, an alternative to manually entering the surface code is to click on the “Surface” button in the “Define” section above the workbook. This button becomes available when the cursor is moved to the surface type column . Doing so brings up the small sub-window shown in Figure 10. The surface type can be selected via the dropdown box at the top of the window. As different types are selected, a graphical description appears complete with the appropriate set of input parameter boxes on the right side of the window. Entering the correct parameters and clicking on the “OK” button results in the correct code appearing as the column entry.

Specifying The Optical Interaction

* Interaction options - reflection / refraction
* Choosing the optics

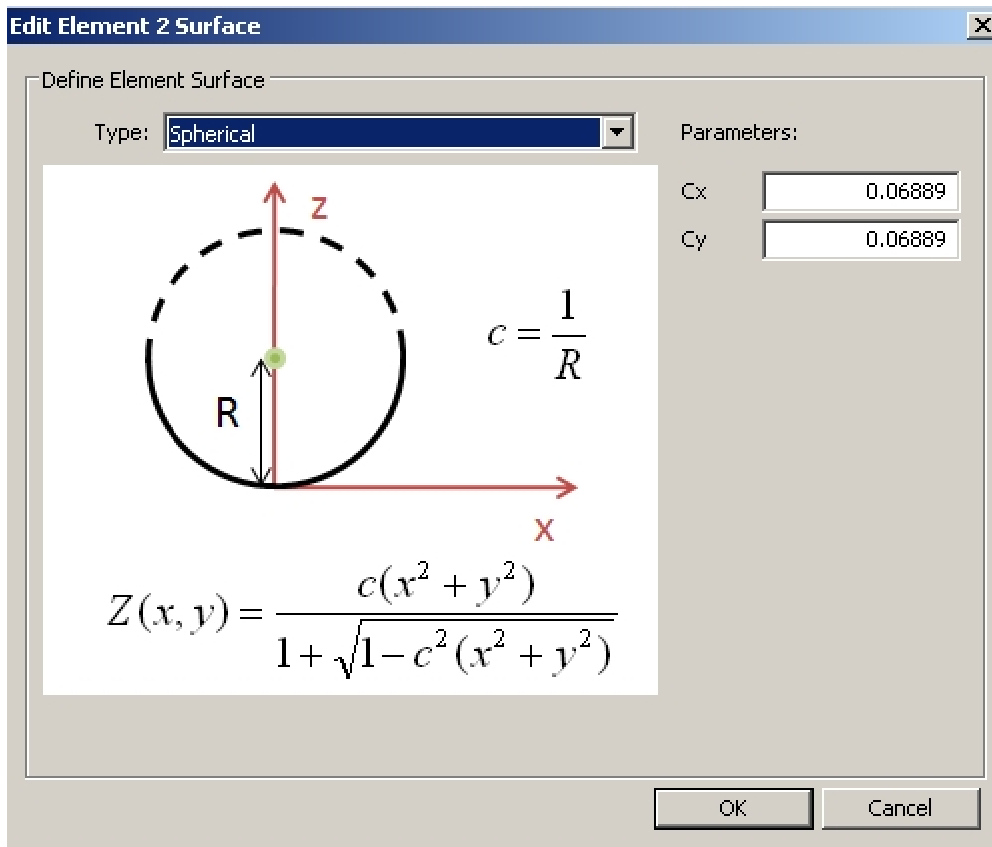


Figure 10. Surface definition window

Optical Properties

Each element can have a distinct set of optical parameters that define how rays interact with the surface. In the "Interaction" column, the user can select between Refraction and Reflection. Other types of interactions may become available in the future, including Aperture Stop, Diffraction via Transmission, and Diffraction via Reflection. Currently, only the reflective and refractive interaction types have been validated, even though parameters for the other optic types are available on the optical property definition window.

Optical Interaction Type	*Code*
Refraction	1
Reflection	2
Aperture Stop	3
Diffraction via Transmission	4
Diffraction via Reflection	5

Table 3. Optical interaction types and corresponding numeric codes

The optical properties are referenced by name. A SolTrace project may have many sets of optical property defined each with a unique name. These are defined on the "Define Optics" input page, and are discussed in the next section in the manual. Once the appropriate property sets are defined, the "Optics" column on the element table can be filled in with the name of the optical properties for that element. Clicking the "Optics..." button above the table will pop up a small dialog listing all the optical property sets defined in the system, and by selecting one and clicking "OK", the name will be entered into the appropriate table cell.

For virtual stages, the interaction type and optical properties are ignored and do not need to be initialized.

Created with the Personal Edition of HelpNDoc: Full-featured Help generator

Optical Properties

Optical properties are defined on the second input page, shown below. Each SolTrace project may have numerous optical property sets defined. Use the Add and Remove buttons to add new sets of optical properties. Each optical property set contains a separate set of parameters for the front and back of the surface, accessed by the corresponding tabs. Optical property sets can be saved and retrieved later using the Import and Export buttons.

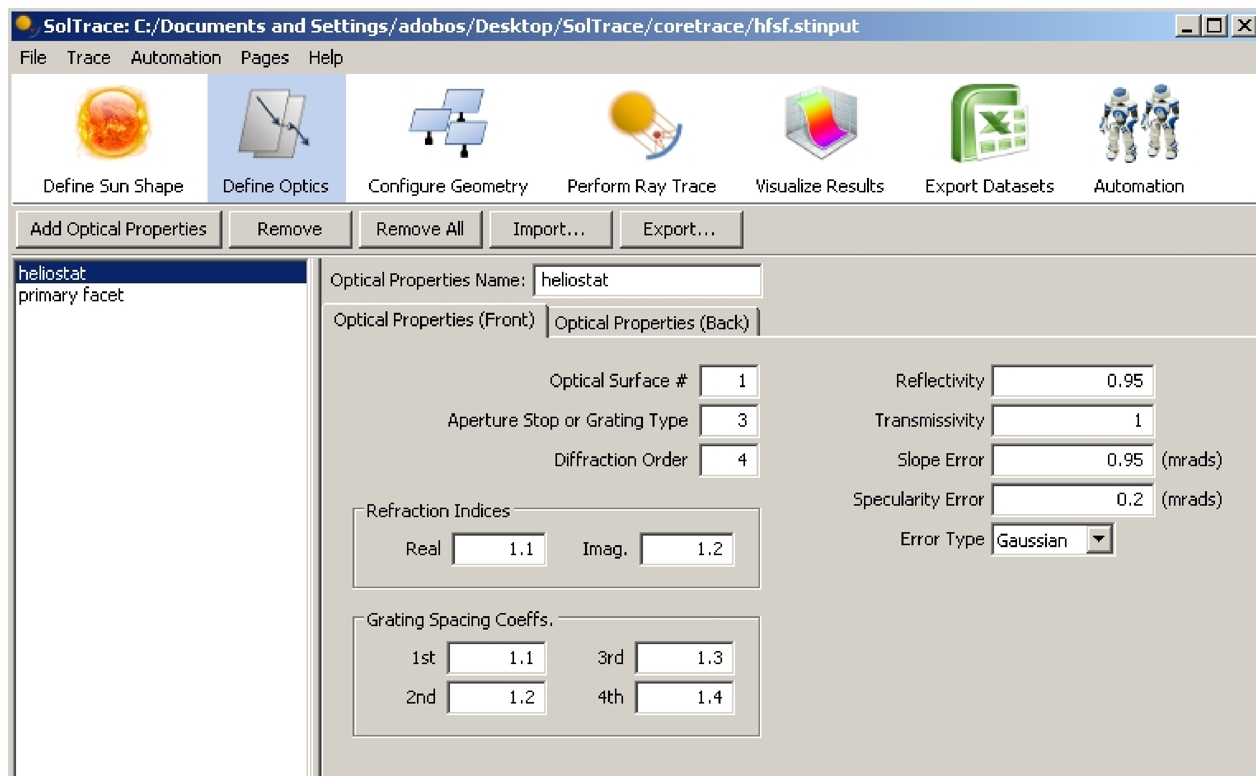


Figure. Optical property setup input page.

For refractive optics, only the transmissivity and the real component of the refraction indices are relevant and used at this time. The imaginary refractive index will not be discussed. A real physical refractive component is actually constructed from two elements or surfaces. In Figure 13 for example, a plane of glass consists of two surfaces (or elements) separated by the glass media between. A ray passes from one media (air for example) through one surface (or element) to the glass media, is refracted and then passes through the other surface back to the air. The first element would be defined with the index of refraction of air on the back side and the index of refraction of glass on the front side. The second surface would be defined with the index of refraction of glass

on the back side and the index of refraction of air on the front side. Surfaces other than flat would construct lenses.

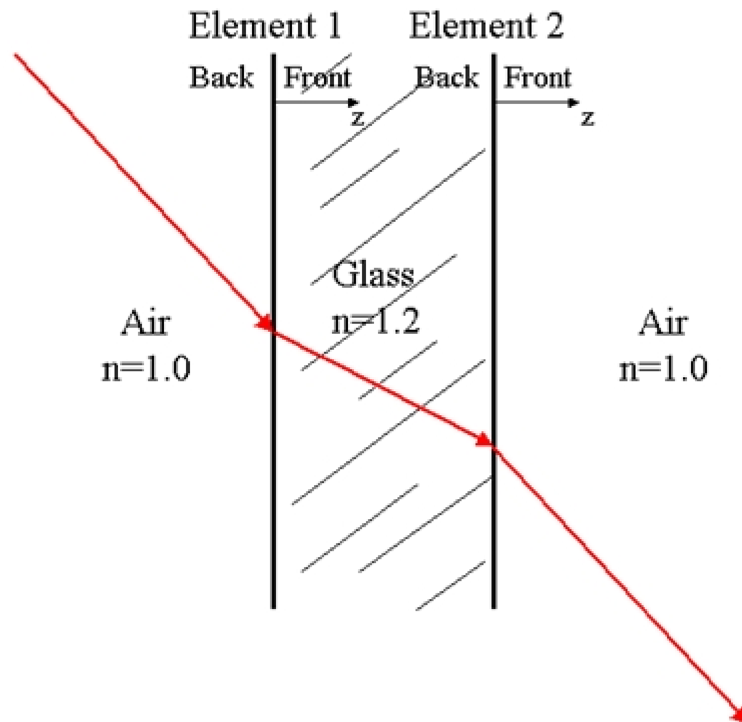


Figure 13. A glass plane is actually constructed from two separate elements.

The transmissivity is the fraction of rays (0 to 1.0) that pass through an element. So, if in the above example the glass plane transmits 98% then one of the elements should be defined with a transmissivity of 0.98 and the other 1.0 or both could be defined with transmissivities of 0.99. Currently, for refractive surfaces there is no reflective component.

For reflective optics, one element is usually sufficient to model a mirror since no transmission is allowed. The relevant parameter is the reflectivity and the element still possesses both back and front side values.

For both refractive and reflective optics another set of optical parameters applies. In addition to the effects of the element surface shape on ray direction, two random errors can be included which affect ray interaction at the surface of an element. They are surface slope error and surface specularity. Both are illustrated in Figure 14 for the case of a reflective surface with Gaussian error distribution. Surface slope error is a macro feature while specularity is a micro structure effect. The total error is given by

$$\sigma_{optical} = (4\sigma_{slope}^2 + \sigma_{specularity}^2)^{1/2}$$

If the pillbox distribution is used than replace σ with the half-width of the pillbox. $\sigma_{optical}$ is in terms of the reflected vector. By Snell's Law a slope error of θ results in a reflected vector error of 2θ. Specularity error is already in terms of the reflected vector. Thus the factor of 4 on the σ_{slope} term.

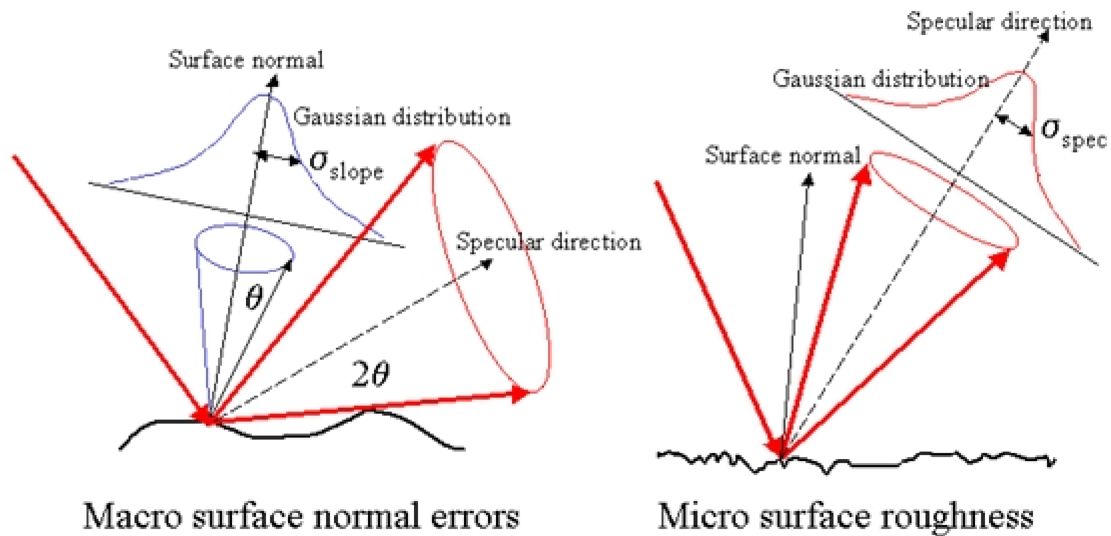


Figure 14. Illustration of surface slope and surface specularity error

Select the distribution type in the box on the lower left of the window and enter appropriate values for the R.M.S. errors to the right. $\sigma_{optical}$ cannot be equal to zero otherwise an error occurs. Therefore, at least one of the two error components should be non-zero. If the user does not wish to include either one of these errors, they can be switched off later before the ray tracing begins. For now, always include some positive value for at least one of these errors.

Created with the Personal Edition of HelpNDoc: Create cross-platform Qt Help files

Tracing

Overview

When the user specifies a certain number of rays to be traced (how this is done will be described later), rays continue to be randomly generated until that number of intersections has occurred somewhere on the elements of Stage 1. A vector is calculated which connects the origin of Stage 1 and the source (e.g. the sun) and ray locations are generated on a plane normal to this vector and within a circle which just encompasses all the projected shapes of the elements within Stage 1. This narrows down the random ray generation region and saves time. The actual application of the sun direction and the sunshape occurs after these intersections have been determined, not before as might be expected. Mathematically it makes no difference when they are applied, but functionally it is more efficient to do so in this way in order to eliminate needless ray generation and, subsequently, time. Once the requested number of rays has intersected Stage 1 somewhere, they are traced to subsequent stages. The 'Trace Through' checkbox thus never appears on the Stage 1 worksheet, because the requested ray number always intersects Stage 1 somewhere.

In certain optical systems, the optical path can fold back on itself. An example would be that of the solar furnace described in Figure 1. A heliostat (Stage 1) reflects sunlight back to a primary concentrator (Stage 2) which in turn focuses and redirects the light back to a target (Stage 3). In this case, Stage 3 could actually lie between Stages 1 and 2. SolTrace does not 'know' this fact. It only knows sequential stage numbers. After the

first stage, when a ray leaves a subsequent stage it is automatically traced to the next stage (if the 'Trace Through' box is checked) regardless of whether the ray hit an element within the stage or missed all the elements completely. Thus, if the 'Trace Through' box was checked for Stage 2 and a ray was traced that actually missed all the elements of Stage 2 it would continue to be traced to the target in Stage 3 rather than exiting the system completely which is the physical reality. Since the last known ray location and direction was in Stage 1, the ray would continue to be traced to Stage 3 and would intersect on the backside of the target in Stage 3 which is not the physical reality. If the "Trace Through' box had been unchecked for Stage 2 then this ray would have been handled correctly. Upon missing all elements of Stage 2, it would have been logged as missed and not traced further.

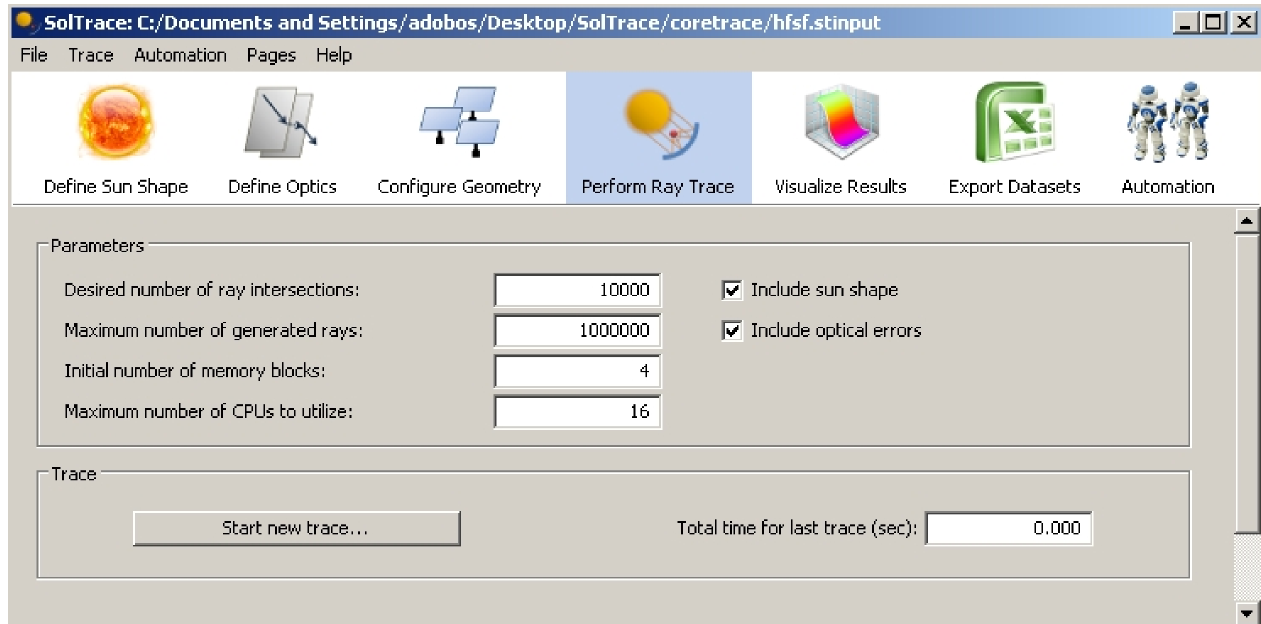


Figure 15. The trace setup parameters

The first input box is the number of rays to be traced. Recall from previous discussions that this many rays will actually interact with elements of Stage 1. The code randomly generates rays on a plane normal to the vector linking the sun with origin of Stage 1. The rays are generated within a circle large enough to just encompass the projected images of all the elements in Stage 1 as seen from the sun. The code continues to generate rays until the requested rays have fallen on elements of Stage 1. The code tallies the total number of rays generated within the circle, both those that fell on an element and those that did not, divides this number into the area of the circle and multiplies the result by the Direct Normal Insolation value in the second input box. This results in a unit power/ray value subsequently used to calculate efficiencies and flux intensities. A seed for the Random Number Generator is internally calculated and is based of a psedorandom computation using the current CPU clock time as a driver.

The maximum number of rays to generate allows the user to limit the maximum number of rays generated, regardless of how many have actually hit Stage 1. This prohibits a trace to spin forever due to an implausibly defined geometry. To avoid hitting this limit in normal operation, it is incumbent upon the user to make sure the number is large enough.

SolTrace allocates memory as it needs it during the trace. Before the trace actually

begins, SolTrace calculates the amount of memory it believes is sufficient to do the trace based on the requested ray number and on the assumption that each ray will hit each stage once. It then does one large allocation of memory. It cannot anticipate the possibility of multiple bounces of a ray or rays within a stage. If SolTrace runs out of memory, it begins to grab memory as it needs it from that point forward. This can drastically reduce execution speed. If the user can anticipate this fact, the number of initial memory blocks can be increased with this spin button thereby minimizing or even eliminating these subsequent memory calls and thus execution time.

For computers with multiple processors, SolTrace automatically dispatches threads to concurrently trace rays. By default, it dispatches as many threads as there are processors in the computer. However, the user may elect to only use one processor or two to leave others free for other tasks, or to asses performance characteristics.

Upon click the "Start new trace" button, a progress dialog will prop up that gives statistics about the current status of the ray tracing procedure. On computers with multiple processors, each processor's progress is shown separately, along with aggregate statistics about the number of rays being traced per second and the total elapsed time.

When the trace has completed, the results are automatically loaded into the visualization and data export modules. Simply click the appropriate toolbar buttons to view graphs or export the data to CSV files or Excel.

Created with the Personal Edition of HelpNDoc: Easy to use tool to create HTML Help files and Help web sites

Visualization

At this time, there are ~~basically three~~ different plot types available within SolTRACE. The 3-D ray intersection and path plot, ~~projected XY, YZ, and XZ scatter plots of intersection points,~~ and a 2-D flux distribution contour plot. Figure 16 is an example of a scatter plot showing ray intersections for a single element Stage 1 consisting of a parabolic mirror and a single element Stage 2 consisting of a flat target at the focus of the parabolic mirror. At the upper right of the plot window, the user will notice two boxes, one entitled 'Stages' and the other entitled 'Elements'. In the Stage box will be a list of the stages within the system. Initially the 'Element' box will be empty. Clicking on (or selecting) one or more of the stages within the stage list displays within the 'Element' box the list of elements for each of the selected stages. In the 'Element' box , the selected stages are shown in gray. A 'V' or an 'O' next to the stage number indicates whether it is a virtual or optical stage type. Below each stage entry is the list of elements within that stage with check boxes next to them. The surface type for each element is also indicated. See Table x for the alphanumeric code designation. The last pieces of information shown in the 'Element' box are the number of intersection points for each stage and for each element within each stage. Note that one ray can have multiple intersections with one or more surfaces. One can select more than one stage if desired resulting in the elements for each stage being displayed in the 'Element' box as shown. At the bottom of the 'Stage' list are two buttons whose purpose is straightforward. One selects all the stages with one click, the other unselects all stages with one click. The user can select ray intersections for specific elements to be plotted by checking the boxes next to the elements of interest. The buttons below this box also perform the same function as the previous two for the stage list.

Once the elements of interest have been selected, the user can now choose how to display the ray intersections. Directly aboce the stage list is a box for selecting the

coordinate system to be used for the plot. If elements from multiple stages have been selected, then clearly the global coordinate system would be the logical choice. However, if a set of elements within one stage is selected, the user may wish to plot the ray intersections in the coordinate system for that stage. The same holds true if only one element was selected, the user may wish to plot the results in the coordinate system for that element. The "Final intersections only" checkbox to the right allows the user to display all ray intersections including multiple bounces or just the absorbed or final ray intersections (intersection before ray exits a stage) within a particular stage. There are three options available to the user for displaying the ray intersections. The first is displaying them all as one series or color. Another is to plot ray intersections at each individual element (regardless of stage) as separate series or colors. The last option is to plot ray intersections within each stage as separate series or colors. Ray paths for individual rays can also be shown as lines connecting ray intersection points (useful for tracking ray paths through complicated multiple bounce geometries). Each ray has a number associated with it from 1 to the number of requested rays. Specific ray numbers can be plotted or a range of ray numbers (e.g. '1,2,5,6-10'). Clicking on 'Ray Numbers:' makes the text box available for entering the ray numbers of interest. Rays paths are shown in black with the exception of rays that miss the last displayed stage. The last leg of a ray path that ultimately misses the last displayed stage is shown in white as it exits.

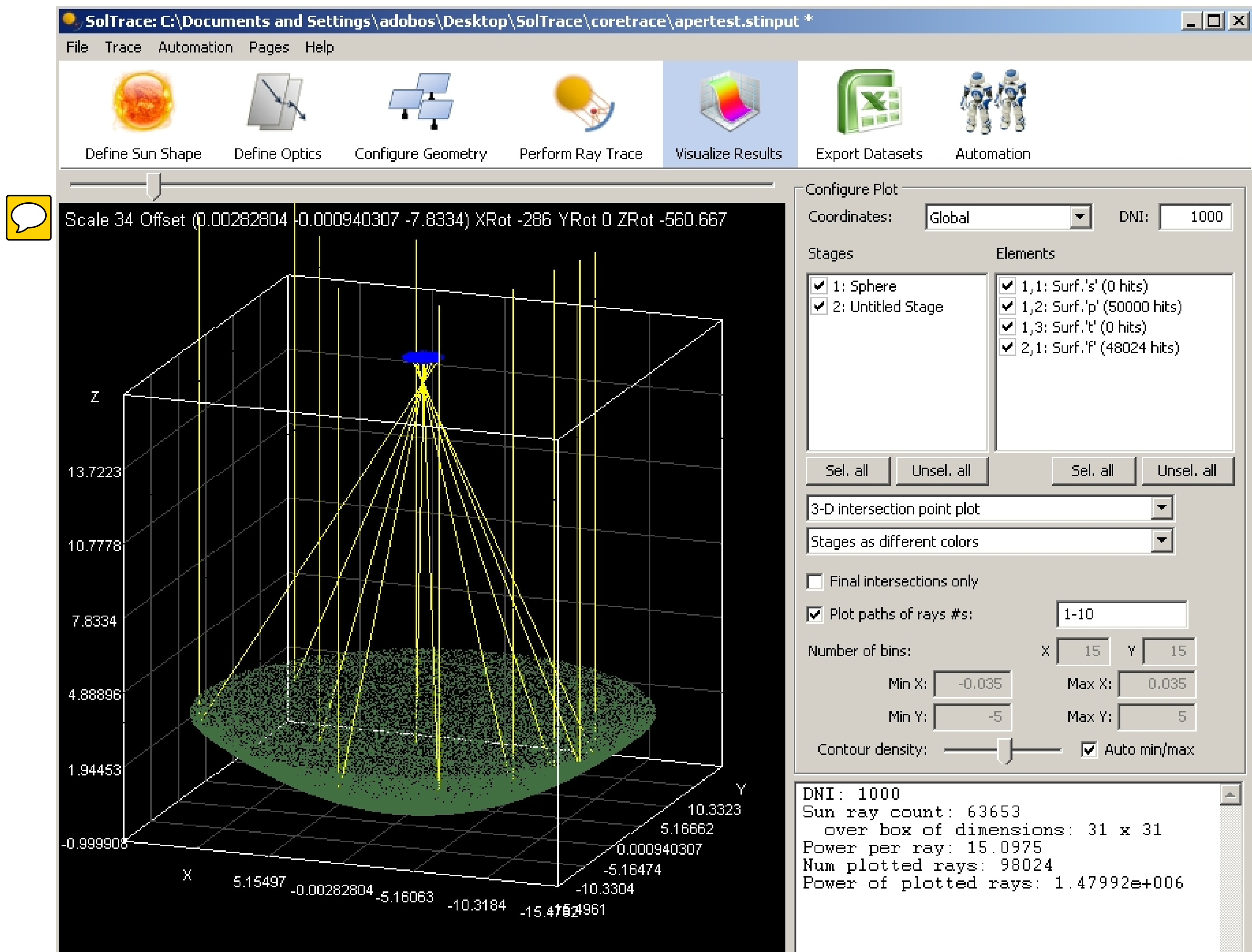


Figure 16

Some statistical information is displayed once the plot is complete. The r.m.s. radius of the distribution from the coordinate system origin and the r.m.s deviation are displayed together with the centroid of the distribution relative to the coordinate system origin.

This information is only meaningful for intersections on a flat plane.

Other plot types can be selected using the dropdown below the stage and element check lists. Both the surface plot and the contour plot are designed for flux distributions on a plane and therefore require that the data be for one element with either a flat or cylindrical surface in the element coordinate system. A cylindrical element is "rolled out flat" for plotting purposes. The program will not allow a surface or contour plot if 1) more than one element is selected, 2) a coordinate system other than the element coordinate system is selected, or 3) the element surface is not flat or cylindrical. The number of bins in the x and y directions must be entered. The program calculates the minimum and maximum extents in both directions and uses these by default for plotting, however the user can enter other minimums and maximums as desired. Below the plotting buttons, some statistical information on the flux distribution is displayed. Peak flux, verage flux and the r.m.s radius of the flux distribution are displayed in both the units chosen by the user and also in suns (units of the direct normal irradiance value entered in the 'Trace' form.)

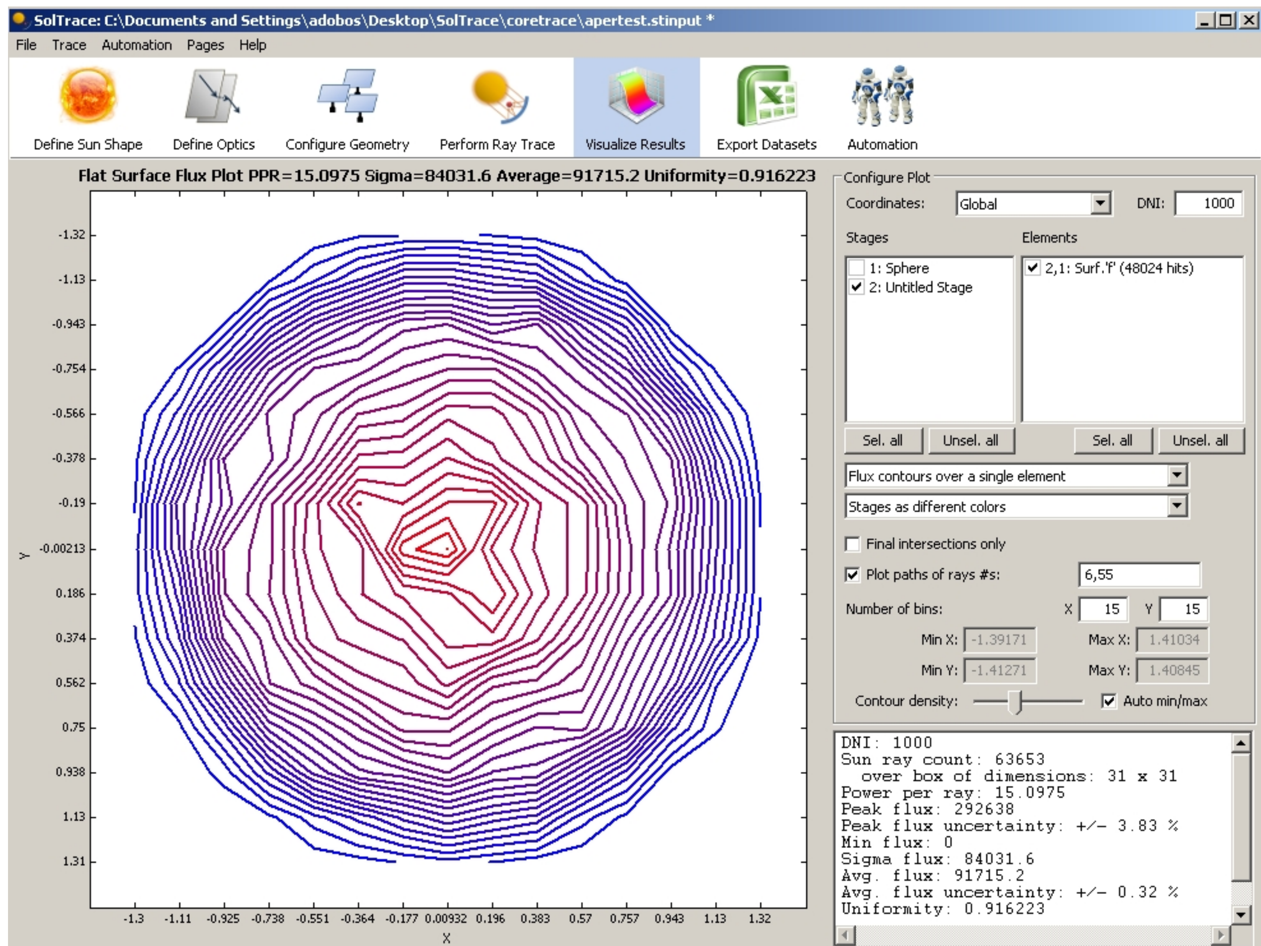
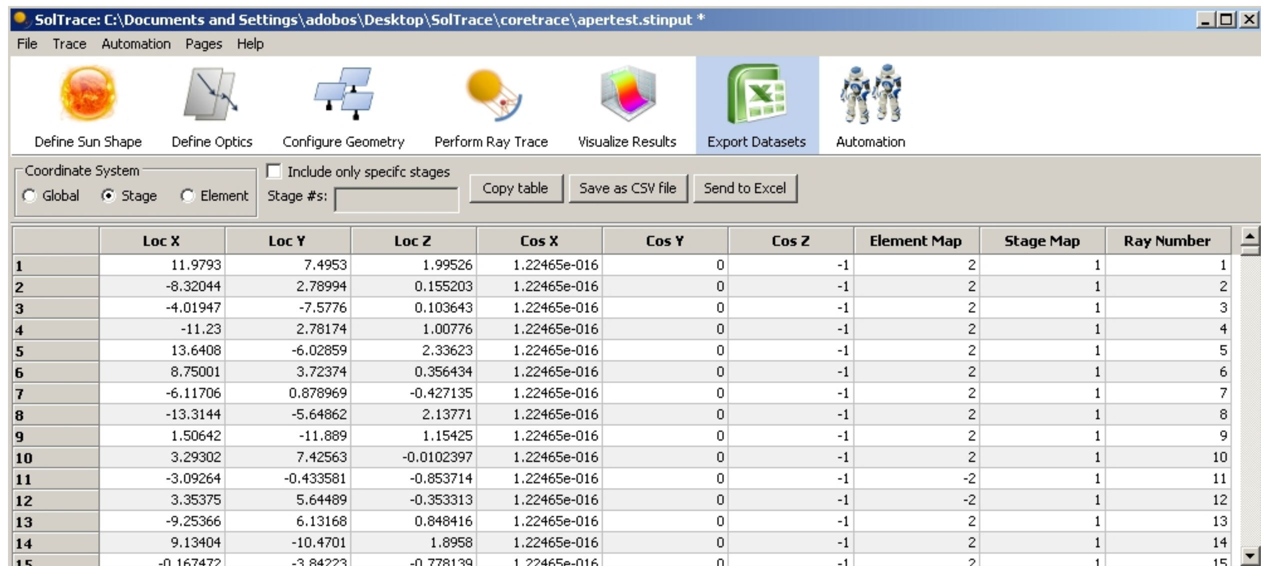


Figure. Flux contour plot with statistics.

Created with the Personal Edition of HelpNDoc: Easily create Web Help sites

Exporting Data

Upon completion of a trace, the raw intersection point data can be viewed and exported on the "Export Data" page.



	Loc X	Loc Y	Loc Z	Cos X	Cos Y	Cos Z	Element Map	Stage Map	Ray Number
1	11.9793	7.4953	1.99526	1.22465e-016	0	-1	2	1	1
2	-8.32044	2.78994	0.155203	1.22465e-016	0	-1	2	1	2
3	-4.01947	-7.5776	0.103643	1.22465e-016	0	-1	2	1	3
4	-11.23	2.78174	1.00776	1.22465e-016	0	-1	2	1	4
5	13.6408	-6.02859	2.33623	1.22465e-016	0	-1	2	1	5
6	8.75001	3.72374	0.356434	1.22465e-016	0	-1	2	1	6
7	-6.11706	0.878969	-0.427135	1.22465e-016	0	-1	2	1	7
8	-13.3144	-5.64862	2.13771	1.22465e-016	0	-1	2	1	8
9	1.50642	-11.889	1.15425	1.22465e-016	0	-1	2	1	9
10	3.29302	7.42563	-0.0102397	1.22465e-016	0	-1	2	1	10
11	-3.09264	-0.433581	-0.853714	1.22465e-016	0	-1	-2	1	11
12	3.35375	5.64489	-0.353313	1.22465e-016	0	-1	-2	1	12
13	-9.25366	6.13168	0.848416	1.22465e-016	0	-1	2	1	13
14	9.13404	-10.4701	1.8958	1.22465e-016	0	-1	2	1	14
15	-0.167472	-3.84223	-0.778139	1.22465e-016	0	-1	2	1	15

Figure 20. Handling raw data points

Tabular data of ray intersections and directions can be loaded into a worksheet by clicking on the button in the upper left corner of the form. The purpose is to provide a quick way to inspect the data for reasonableness and accuracy. The format seen here is the exact way in which the data is saved to a text file. The data shown in the table is what is saved as a CSV file or exported to Excel. So if a given geometry consisted of three stages for example, then three separate files would have to be saved to capture all ray interactions within that geometry. By default, the results for all stages are listed, but the user can limit the display to only certain stages using the checkbox and input above the data table. The coordinate system in which the data is to be cast is selected in the upper right of the form.

Recall that when the user selects a given ray number in the 'Trace' form, that number of rays is generated somewhere on the elements of Stage 1. Each ray is numbered from one to the requested ray number and can be identified and traced through the system by this number. The table shown in Figure 20 is a table of intersections for each ray within a particular stage. Each entry contains: 1) the x,y,z location of the ray intersection point within the stage, cast in whatever coordinate system the user chose, 2) the direction cosines of the ray as it *entered* the stage, 3) the element number within the stage that the ray hit, 4) the associated stage number, and 5) the ray number. If a particular ray had multiple intersections within a stage then there will be multiple entries for that ray number showing the path that it took through that particular stage before exiting. If a ray missed all elements of the stage completely then there will be a zero in the element number column signifying this occurrence. If the 'trace through' for that stage was turned off, then that ray number will no longer show up in any of the following stage data lists as it is considered a lost ray. If the element number is negative for a particular ray intersection, it means that the ray in fact hit this element but was absorbed. Similarly, this ray ceases to be traced though the rest of the system and so will not show up in subsequent stage data lists.

This data is provided so that it can be processed outside of SolTrace with other plotting packages, heat transfer/radiation analysis programs, etc.

Created with the Personal Edition of HelpNDoc: iPhone web sites made easy

SolScript Automation

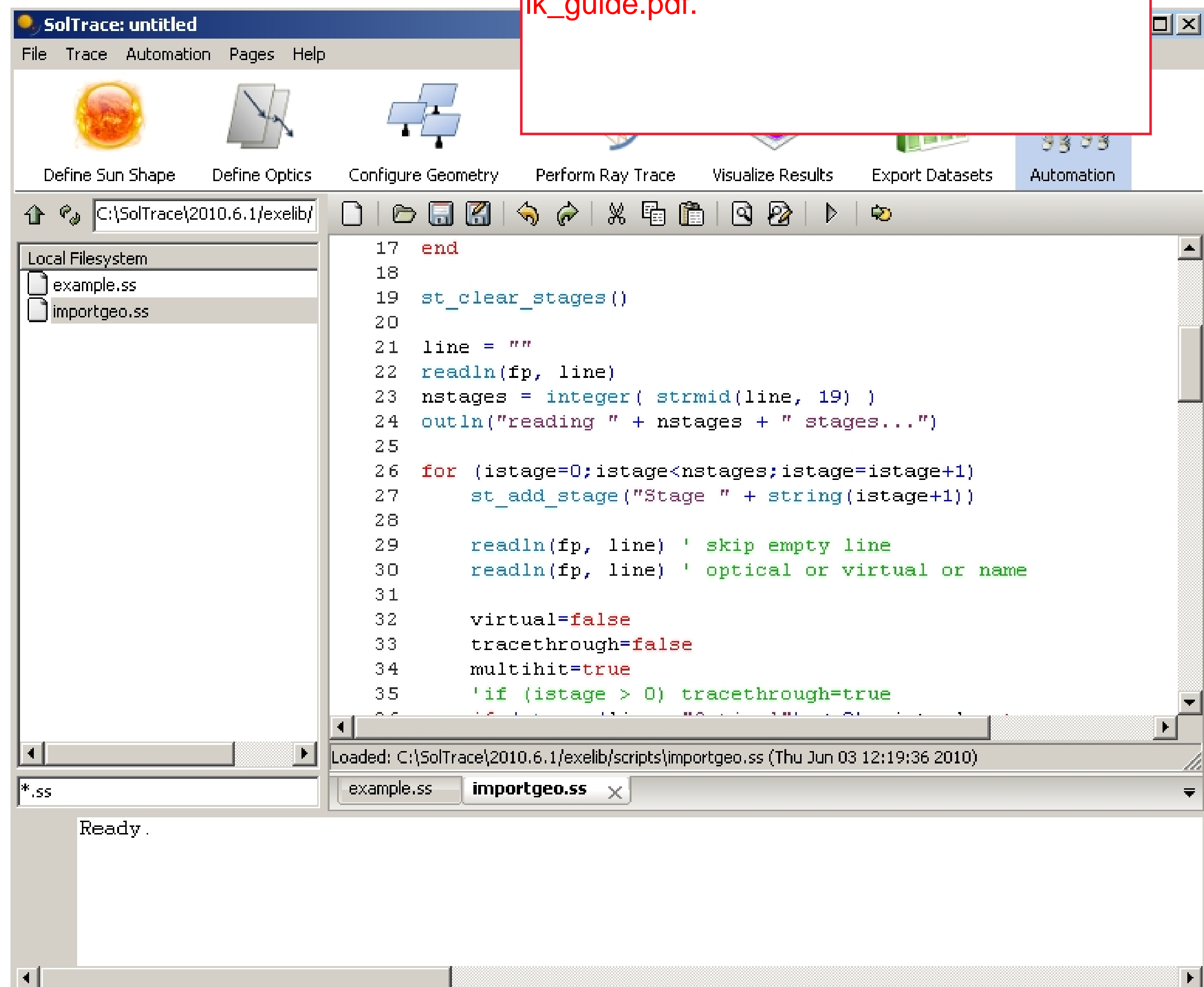
The SolTrace automation capabilities use the simple Scriptlet language engine. This is the same scripting language used by the Solar Advisor Model (SAM) tool, although several built-in functions are provided for working directly with SolTrace projects. associated working environment will be refer

This entire section is out of date. SolTrace now uses LK scripting with custom functions. The routines are documented in the script window function reference. To update this section, add description of how to create/ open an LK file, then add a listing of the available functions and their documentation. The listing can pretty much be pulled directly from the function reference window in LK. We can also remove a lot of the background material on language usage and syntax, and refer users to the LK documentation at https:// github.com/NREL/lk/blob/develop/doc/ lk_guide.pdf.

1.1 Entering a SolScript Progran

SolScript is included by default in every insta on' tab on the top navigation toolbar.

SolScript files are not attached to or saved w e text files with the *.ss extension. The automat ting and opening script files, and a multi-tab edito once. The text area below the editor window e script are displayed. A screenshot of the env



To run a script, press the forward arrow button on the toolbar above the editor. This will cause the

script in the active text editor to be loaded and interpreted by the Scriptlet engine. If there are any syntax errors in the program, SolTrace will list the errors in the message window and require the user to fix them before proceeding. If you have typed something incorrectly, the first error location displayed is nearest the point in the source code where the typo occured.

If your code is syntactically correct, SolTrace will run it and display and send any output that your script may create to the messages window.

1.2 Hello world!

As with any new language, the traditional first program is to print the words "Hello, world!" on the screen, and the SolScript version is listed below.

```
out( "Hello, world!\n" )
```

Notable features:

1. The `out` command generates output from the script
2. The text to output is enclosed in double-quotes
3. To move to a new line, the symbol `\n` is used

To run Hello world, type it into a new SolScript script, and press the 'Run' button on the toolbar. Now we will learn some more about variables and program control in SolScript.

1.3 Why SolScript instead of VBA?

Since SolScript is so similar in functionality and syntax to Visual Basic, you might wonder why we didn't simply use a VBA language engine. Visual Basic for Applications (VBA) is a closed source Microsoft product that is very tightly integrated into the Office suite of applications, and while there are some freely available interpreters for subsets of VB-like languages, we decided that they would not suit our purposes well enough.

SolScript is close enough to VBA in general syntax and program structure to make it easily understandable by people familiar with VBA, but with some differences and additional functionality to alleviate some of the more annoying aspects of VBA programming. By developing our own script engine, we have been able to integrate it very tightly into the SolTrace environment for maximum ease of use.

Some notable points of departure from VBA include:

1. '`for`' loop syntax follows the C / Perl convention
2. Array arithmetic is automatically performed (array * array results in element by element multiplication)
3. '`elseif`' statement formatting is similar to PHP
4. No distinction between functions and procedures

Created with the Personal Edition of HelpNDoc: Generate Kindle eBooks with ease

Data Variables

2.1 General Syntax

SolScript is a programming language that is similar in style to Visual Basic. Each program statement is generally placed on a line by itself, and the end-of-line marks the end of the statement. However, unlike Visual Basic, there is no facility to split a long statement across multiple lines.

Blank lines may be inserted between statements. While they have no meaning, they can help make a script easier to read. Spaces can also be added or removed nearly anywhere, except for in the middle of a word. The following statements all have the same meaning.

```
out("Hello 1\n")
  out  ("Hello 2\n")
out (  "Hello 3\n"  )
```

Comments are lines in the program code that are ignored by SolScript. They serve as a form of documentation, and can help other people (and you!) more easily understand what the script does. Comments begin with the single-quote `'` character, and continue to the end of the line.

```
' this program creates a greeting
out( "Hello, world!\n" ) ' display the greeting to the user
```

2.2 Variables

Variables store information while your script is running. SolScript variables share many characteristics with other computer languages.

1. Each variable stores a single value
2. Variables do not need to be "declared" in advance of being used
3. There is no distinction between variables that store text and variables that store numbers

Variable names may contain letters, digit, and the underscore symbol. A limitation is that variables cannot start with a digit. Unlike some languages like C and Perl, SolScript does not distinguish between upper and lower case letters in a variable (or subroutine) name. As a result, the name `myData` is the same as `MYdata`.

Values are assigned to variables using the equal sign `=`. Some examples are below.

```
Num_Modules = 10
ArrayPowerWatts = 4k
Tilt = 18.2
system_name = "Super PV System"
Cost = "unknown"
COST = 1e6
cost = 1M
```

A second assignment to a variable erases its previous value. As shown above, decimal numbers can be written using scientific notation or engineering suffixes. The last two assignments to `Cost` are the same value. Recognized suffixes are listed in the table below. Note that suffixes are case-sensitive.

Name	Suffix	Multiplier
Tera	T	1e12
Giga	G	1e9
Mega	M	1e6
Kilo	k	1e3
Milli	m	1e-3
Micro	u	1e-6
Nano	n	1e-9
Pico	p	1e-12
Femto	f	1e-15
Atto	a	1e-18

Table 1: Recognized Engineering Suffixes

2.3 Arithmetic

SolScript supports the four basic operations `+`, `-`, `*`, and `/`. The usual algebraic precendence rules are followed, so that multiplications are performed before additions. Parentheses are also understood and can be used to change the default order of operations.

More complicated operations like raising to a power and performing modulus arithmetic are possible using built-in function calls in the standard SolScript library.

Examples of arithmetic operations:

```
battery_cost = cost_per_kwh * battery_capacity

' multiplication takes precedence
degraded_output = degraded_output - degraded_output * 0.1

' use parentheses to subtract before multiplication
cash_amount = total_cost * ( 1 - debt_fraction/100.0 )
```

2.4 Simple Input and Output

You can use the built-in `out` and `outln` functions to write data to the console window. The difference is that `outln` automatically appends a newline character to the output. To output multiple text strings or variables, use the `+` operator, or separate them with a comma.

```
array_power = 4.3k
array_eff = 0.11
outln("Array power is " + array_power + " Watts.")
outln("It is " + (array_eff*100) + " percent efficient.")
outln("It is ", array_eff*100, " percent efficient.") ' same as above
```

The console output generated is:

```
Array power is 4300 Watts.
It is 11 percent efficient.
```

Use the `in` function to read input from the user. You can optionally pass a message to `in` to display to the user when the input popup appears. The user can enter either numbers or text, and SolScript will perform any type conversions if needed (and if possible).

```
cost_per_watt = in("Enter cost per watt:")  ' Show a message.  in() also is
fine.
notice( "Total cost is: " + cost_per_watt * array_power + " dollars")
```

The `notice` function works like `out`, except that it displays a popup message box on the computer screen.

2.5 Data Types and Conversion

SolScript supports four basic types of data, although most conversions between types happen automatically.

Type	Conversion Function	Valid Values
Integer Number	`integer()`	+/- approx. 2 billion
Double-precision Decimal Number	`double()`	1e-308 to 1e308, with infinity
Boolean	`boolean()`	`true` or `false` (`1` or `0`)
Text Strings	`string()`	Any length text string

Table 2: Intrinsic SamUL Data Types

Sometimes you have two numbers in text strings that you would like to multiply. This can happen if you read data in from a text file on the computer, for example. Since it does not make sense to try to multiply text strings, you need to first convert the strings to numbers. To convert a variable to a double-precision decimal number, use the `double` function, as below.

```
a = "3.5"
b = "-2"
c1 = a*b ' this will cause an error when you click 'Run'
c2 = Double(a) * Double(b) ' this will assign c2 the number value of -7
```

You can also use `integer` to convert a string to an integer or truncate a decimal number, or the `string` function to explicitly convert a number to a string variable.

If you need to find out what type a variable currently has, use the `typeof` function to get a description.

```
a = 3.5
b = -2
c1 = a+b ' this will set c1 to -1.5
c2 = String( Integer(a) ) + String( b ) ' c2 set to text "3-2"
```

```
outln( typeof(a) ) ' will display "double"
outln( typeof(c2) ) ' will display "string"
```

2.6 Special Characters

Text data can contain special characters to denote tabs, line endings, and other useful elements that are not part of the normal alphabet. These are inserted into quoted text strings with *escape sequences*, which begin with the \ character.

Escape Sequence	Meaning
`\n`	New line
`\t`	Tab character
`\r`	Carriage return
`\"`	Double quote
`\\`	Backslash character

Table 3: Text String Escape Sequences

So, to print the text `"Hi, tabbed world!"`, or assign `c:\Windows\notepad.exe`, you would have to write:

```
outln("\"Hi,\ttabbed world!\"")
program = "c:\\Windows\\notepad.exe"
```

Created with the Personal Edition of HelpNDoc: Produce online help for Qt applications

Flow Control

3.1 Comparison Operators

SolScript supports many ways of comparing data. These types of tests can control the program flow with branching and looping constructs that we will discuss later.

There are six standard comparison operators that can be used on most types of data. For text strings, "less than" and "greater than" are with respect to alphabetical order.

Comparison	Operator
Equal	`==`
Not Equal	`!=`
Less Than	`<`
Less Than or Equal	`<=`

Greater Than	>
Greater Than or Equal	>=

Table 4: Comparison Operators

Examples of comparisons:

```
divisor != 0
state == "oregon"
error <= -0.003
"pv" > "csp"
```

Single comparisons can be combined by *boolean* operators into more complicated tests.

1. The `not` operator yields true when the test is false. It is placed before the test whose result is to be notted.
   Example: `not (divisor == 0)`

2. The `and` operator yields true only if both tests are true.
   Example: `divisor != 0 and dividend > 1`

3. The `or` operator yields true if either test is true.
   Example: `state == öregon" or state == "colorado"`

The boolean operators can be combined to make even more complex tests. The operators are listed above in order of highest precedence to lowest. If you are unsure of which test will be evaluated first, use parentheses to group tests. Note that the following statements have very different meanings.

```
state_count > 0 and state_abbrev == "CA" or state_abbrev == "OR"
state_count > 0 and (state_abbrev == "CA" or state_abbrev == "OR")
```

3.2 Branching

Using the comparison and boolean operators to define tests, you can control whether a section of code in your script will be executed or not. Therefore, the script can make decisions depending on different circumstances and user inputs.

3.2.1 `if` Statements

The simplest branching construct is the `if` statement. For example:

```
if ( tilt < 0.0 )
    outln("Error: tilt angle must be 0 or greater")
end
```

Note the following characteristics of the `if` statement:

1. The test condition is placed in parentheses after the `if` keyword.

2. The following program lines include the statements to execute when the `if` test succeeds.

3. For the sake of program readability, the statements inside the `if` are indented. The program will still be correct if they are not indented, but then the script becomes much harder to understand

and debug.

4. The construct concludes with the `end` keyword.
5. When the `if` test fails, the program statements inside the `if-end` block are skipped.

3.2.2 `else` Construct

When you also have commands you wish to execute when the `if` test fails, use the `else` clause. For example:

```
if ( power > 0 )
    energy = power * time
    operating_cost = energy * energy_cost
else
    outln("Error, no power was generated.")
    energy = -1
    operating_cost = -1
end
```

3.2.3 Multiple `if` Tests

Sometimes you wish to test many conditions in a sequence, and take appropriate action depending on which test is successful. In this situation, use the `elseif` clause. Be careful to spell it as a single word, as both `else if` and `elseif` can be syntactically correct, but have different meanings.

```
if ( angle >= 0 and angle < 90)
    text = "first quadrant"
elseif ( angle >= 90 and angle < 180 )
    text = "second quadrant"
elseif ( angle >= 180 and angle < 270 )
    text = "third quadrant"
else
    text = "fourth quadrant"
end
```

You do not need to end a sequence of `elseif` statements with the `else` clause, although in most cases it is appropriate so that every situation can be handled. You can also nest `if` constructs if needed. Again, we recommend indenting each "level" of nesting to improve your script's readability. For example:

```
if ( angle >= 0 and angle < 90 )
    if ( print_value == true )
        outln( "first quadrant: " + angle )
    else
        outln( "first quadrant" )
    end
end
```

3.2.4 Single line `ifs`

Sometimes you only want to take a single action when an `if` statement succeeds. To reduce the amount of code you must type, SolScript accepts single line `if` statements, as shown below.

```
if ( azimuth < 0 ) outln( "Warning: azimuth < 0, continuing..." )

if ( tilt > 90 ) tilt = 90   ' set maximum tilt value
```

You can also use an `else` statement on single line `if`. Like the `if`, it only accepts one program statement, and must be typed on the same program line. Example:

```
if ( value > average ) outln("Above average") else outln("Not above average")
```

3.3 Looping

A loop is a way of repeating the same commands over and over. You may need to process each line of a file in the same way, or sort a list of names. To achieve such tasks, SolScript provides two types of loop constructs, the `while` and `for` loops.

Like `if` statements, loops contain a "body" of program statements followed by the `end` keyword to denote where the loop construct ends.

3.3.1 `while` Loops

The `while` loop is the simplest loop. It repeats one or more program statements as long as a logical test holds true. When the test fails, the loop ends, and the program continues execution of the statements following the loop construct. For example:

```
while ( done == false )
    ' process some data
    ' check if we are finished and update the 'done' variable
end
```

The test in a `while` loop is checked before the body of the loop is entered for the first time. In the example above, we must set the variable `done` to `false` before the loop, because otherwise no data processing would occur. After each iteration ends, the test is checked again to determine whether to continue the loop or not.

3.3.2 Counter-driven Loops

Counter-driven loops are useful when you want to run a sequence of commands for a certain number of times. As an example, you may wish to display only the first 10 lines in a text file.

There are four basic parts of implementing a counter-driven loop:

1. Initialize a counter variable before the loop begins.
2. Test to see if the counter variable has reached a set maximum value.
3. Execute the program statements in the loop, if the counter has not reached the maximum value.
4. Increment the counter by some value.

For example, we can implement a counter-driven loop using the `while` construct:

```
i = 0      ' use i as counter variable
while (i < 10)
    outln( "value of i is " + i )
    i = i + 1
end
```

3.3.3 `for` Loops

The `for` loop provides a streamlined way to write a counter-driven loop. It combines the counter

initialization, test, and increment statements into a single line. The script below produces exactly the same effect as the `while` loop example above.

```
for ( i = 0; i < 10; i = i+1 )
    outln( "value of i is " + i )
end
```

The three loop control statements are separated by semicolons in the `for` loop statement. The initialization statement (first) is run only once before the loop starts. The test statement (second) is run before entering an iteration of the loop body. Finally, the increment statement is run after each completed iteration, and before the test is rechecked. Note that you can use any assignment or calculation in the increment statement.

Just like the `if` statement, SolScript allows `for` loops that contain only one program statement in the body to be written on one line. For example:

```
for ( val=57; val > 1; val = val / 2 ) outln("Value is " + val )
```

3.3.4 Loop Control Statements

In some cases you may want to end a loop prematurely. Suppose under normal conditions, you would iterate 10 times, but because of some rare circumstance, you must break the loop's normal path of execution after the third iteration. To do this, use the `break` statement.

```
value = double( in("Enter a starting value") )
for ( i=0; i<10; i=i+1 )
    outln("Value is " + value )
    if (value < 0)
        break
    end
    value = value / 3.0
end
```

In another situation, you may not want to altogether break the loop, but skip the rest of program statements left in the current iteration. For example, you may be processing a list of files, but each one is only processed if it starts with a specific line. The `continue` keyword provides this functionality.

```
for ( i=0; i<file_count; i=i+1 )
    file_header_ok = false

    ' check if whether current file has the correct header

    if (file_header_ok == false)
        continue
    end

    ' process this file
end
```

The `break` and `continue` statements can be used with both `for` and `while` loops. If you have nested loops, the statements will act in relation to the nearest loop structure. In other words, a `break` statement in the body of the inner-most loop will only break the execution of the inner-most loop.

3.4 Quitting

SolScript script execution normally ends when there are no more program statements to run at the end

of the program. However, sometimes you may need to halt early, if the user chooses not to continue an operation, for example.

The `exit` statement will end the SolScript script immediately. For example:

```
if ( yesno("Do you want to quit?") == true )
    outln("Aborted.")
    exit
end
```

The `yesno` function call displays a message box on the user's screen with yes and no buttons, showing the given message. It returns `true` if the user clicked yes, or `false` otherwise.

Created with the Personal Edition of HelpNDoc: Easy CHM and documentation editor

Arrays of Data

Often you need to store a list of related values. For example, you may need to refer to the price of energy in different years. Or you might have a table of state names and capital cities. In SolScript, you can use arrays to store these types of collections of data.

4.1 Initializing and Indexing

An *array* is simply a list of variables that are indexed by numbers. Each variable in the array is called an *element* of the array, and the position of the element within the array is called the element's *index*. The index of the first element in an array is always 0.

To access array elements, enclose the index number in square brackets immediately following the variable name. Unlike many computer languages, SolScript does not require you to declare or allocate space for the array data in advance.

```
names[0] = "Sean"
names[1] = "Walter"
names[2] = "Pam"
names[3] = "Claire"
names[4] = "Patrick"

outln( names[3] ) ' output is "Patrick"
my_index = 2
outln( names[my_index] ) ' output is "Pam"
```

You can also initialize a fixed array using the `array` command provided in SamUL. Simply separate each element with a comma. There is no limit to the number of elements you can pass to `array`.

```
names = array("Sean", "Walter", "Pam", "Claire", "Patrick")
outln( "First: " + names[0] )
outln( "All: " + names )
```

Note that calling the `typeof` function on an array variable will return ärray" as the type description, not the type of the elements. This is because SolScript is not strict about the types of variables stored in an array, and does not require all elements to be of the same type.

4.2 Array Length

Sometimes you do not know in advance how many elements are in an array. This can happen if you are

reading a list of numbers from a text file, storing each as an element in an array. After the all the data has been read, you can use the `length` function to determine how many elements the array contains.

```
count = length( names )
```

4.3 Processing Arrays

Arrays and loops naturally go together, since frequently you may want to perform the same operation on each element of an array. For example, you may want to find the total sum of an array of numbers.

```
numbers = array( 1, -3, 2.4, 9, 7, 22, -2.1, 5.8 )

count = length( numbers )
sum = 0
for (i=0; i<count; i=i+1)
    sum = sum + numbers[i]
end
```

The important feature of this code is that it will work regardless of how many elements are in the array.

4.4 Multidimensional Arrays

As previously noted, SolScript is not strict with the types of elements stored in an array. Therefore, a single array element can even be another array. This allows you to define matrices with both row and column indexes, and even three dimensional arrays.

To create a multi-dimensional array, simply separate the indices with commas between the square brackets. For example:

```
data[0,0] = 3
data[0,1] = -2
data[1,0] = 5
data[2,0] = 1

nrows = length(data) ' result is 4
ncols = length(data[0]) ' result is 2

row1 = data[0]  ' extract the first row

x = row1[0] ' value is 3
y = row1[1] ' value is -2
```

4.5 Managing Array Storage

When you define an array, SolScript automatically allocates sufficient computer memory to store the elements. If you know in advance that your array will contain 100 elements, for example, it can be much faster to allocate the computer memory before filling the array with data. Use the `allocate` command to make space for 1 or 2 dimensional arrays.

```
data = allocate(3,2)  ' a matrix with 3 rows and 2 columns
data[2,1] = 3

prices = allocate( 5 ) ' a simple 5 element array
```

As before, you can extend the array simply by using higher indexes. However, if you know in advance how many more elements you will be adding, it can be faster to use the `resize` command to reallocate

computer memory to store the array. `resize` preserves any data in the array, or truncates data if the new size is smaller than the old size.

```
data = allocate(5)
outln( length(data) )
resize(data, 10)
outln( length(data) )

resize(data, 2, 4)
outln( length(data) )
outln( length( data[0] ) )
```

Created with the Personal Edition of HelpNDoc: Easily create Help documents

Function Calls

It is usually good progamming practice to split a larger program up into smaller sections, often called procedures, functions, or subroutines. A program may be easier to read and debug if it is not all thrown together, and you may have common blocks of code that appear several times in the program.

5.1 User Functions

A function is simply a named chunk of code that may be called from other parts of the script. It usually performs a well-defined operation on a set of variables, and it may return a computed value to the caller.

Functions can be written anywhere in your script, including after they are called. If a function is never called by the program, it has no effect.

5.1.1 Definition

Consider the very simple procedure listed below.

```
function show_welcome()
   outln("Thank you for choosing SolTrace.")
   outln("This text will only be displayed at the start of the script.")
end
```

Notable features:

1. Use the `function` keyword to define a new function.
2. The name immediately follows. Valid function names can have letters, digits, and underscores, but cannot start with a digit.
3. The empty parentheses after the name indicate that this function takes no parameters.
4. The `end` keyword closes the function definition.

To call the function from elsewhere in the code, simply write the function's name, followed by the parentheses.

```
' show a message to the user
show_welcome()
```

5.1.2 Returning a Value

A function is generally more useful if it can return information back to the program that called it. In this example, the function will not return unless the user enters ÿes" or "no" into the input dialog.

```
function require_yes_or_no()
   while( true )
      answer = in("Destroy everything? Enter yes or no:")
      if (answer == "yes") return true
      if (answer == "no") return false
      outln("That was not an acceptable response.")
   end
end

' call the input function
result = require_yes_or_no() ' returns true or false
if ( not result )
   outln("user said no, phew!")
   exit
else
   outln("destroying everything...")
end
```

The important lesson here is that the main script does not worry about the details of how the user is questioned, and only knows that it will receive a `true` or `false` response. Also, the function can be reused in different parts of the program, and each time the user will be treated in a familiar way.

5.1.3 Parameters

In most cases, a function will accept arguments when it is called. That way, the function can change its behavior, or take different inputs in calculating a result. Analagous to mathematical functions, SolScript functions can take arguments to compute a result that can be returned. Arguments to a function are given names and are listed between the parentheses on the function definition line.

For example, consider a function to determine the minimum of two numbers:

```
function minimum(a, b)
   if (a < b) return a else return b
end

' call the function
count = 129
outln("Minimum: " + minimum( count, 77))
```

In SolScript, changing the value of a function's named arguments will modify the variable in the calling program. Instead of passing the actual value of a parameter `a`, SolScript always passes a *reference* to the variable in the original program. The reference is hidden from the user, so the variable acts just like any other variable inside the function.

Because arguments are passed by reference (as in Fortran, for example), a function can "return" more than one value. For example:

```
function sumdiffmult(s, d, a, b)
   s = a+b
   d = a-b
   return a*b
end
```

```
sum = -1
diff = -1
mult = sumdiffmult(sum, diff, 20, 7)

outln("Sum: " + sum + " Diff: " + diff + " Mult: " + mult) ' will output 27,
13, and 140
```

5.1.4 Variable Scope

Generally, variables used inside a function are considered "local", and cannot be accessed from the caller program. For example:

```
function triple(x)
   y = 3*x
end

triple( 4 )
outln( y ) ' this will fail because y is local to the triple function
```

As we have seen, we can write useful functions using arguments and return values to pass data into and out of functions. However, sometimes there are some many inputs to a function that it becomes very cumbersome to list them all as arguments. Alternatively, you might have some variables that are used throughout your program, or are considered reference values or constants. For these situations, you can define variables to be `global` in SamUL, and then they can be used inside functions and in the main program. For example:

```
global pi = 3.1415926

function circumference( r )
   return 2*pi*r
end

function deg2rad( x )
   return pi/180*x
end

outln( "PI: " + pi )
outln( "CIRC: " + circumference( 3 ) )
outln( "D2R: " + deg2rad( 180 ) )
```

Common programming advice is to minimize the number of global variables used in a program. Sometimes the are certainly necessary, but too many can lead to mistakes that are harder to debug and correct, and can reduce the readability and maintainability of your script.

5.2 Built-in SolScript Functions

Throughout this guide, we have made use of built-in functions like `in`, `outln`, and others. These functions are included with SolScript automatically, and called in exactly the same way as user functions. Like user functions, they can return values, and sometimes they modify the arguments sent to them. Refer to the "Standard Library" section at the end of this guide for documentation on each function's capabilities, parameters, and return values.

Created with the Personal Edition of HelpNDoc: Easy EPub and documentation editor

Input, Output, and System Access

SolScript provides a variety of standard library functions to work with files, directories, and interact with other programs. So far, we have used the `in`, `out`, and `outln` functions to accept user input and display program output in the runtime console window. Now we will learn about accessing files and other programs.

6.1 Working with Text Files

To write data to a text file, use the `writetextfile` function. `writetextfile` accepts any type of variable, but most frequently you will write text stored in a string variable. For example:

```
data = ""
for (i=0;i<10;i=i+1) data = data + "Text Data Line " + string(i) + "\n"
ok = writetextfile( "C:/test.txt", data )
if (not ok) outln("Error writing text file.")
```

Reading a text file is just as simple with the `readtextfile` function.

```
mytext = ""
if (not readtextfile( "C:/test.txt", mytext ))
   outln("could not read text file.")
else
   outln("text data:")
   out(mytext)
end
```

While these functions offer an easy way to read an entire text file, often it is useful to be able to access it line by line. SolScript provides many additional functions for working with files. Use the `open` function to open a file.

```
f = open("C:/test.txt", "r")
line = ""
while ( not eof(f) )
   readln(f, line)
   outln( "My Text Line='" + line + "'" )
end

close( f )
```

In the example above, the second parameter to `open` specifies whether to open the file for reading ("r"), writing ("w"), or appending ("a"). Then, we keep reading lines from the file denoted by `f` until we have reached the end of the file. This is determined by calling the `eof` function in the `while` loop condition. Each line is read by the `readln` function and the `line` variable is filled in with the current line of text. To write to a file, use the corresponding `writeln` function.

Another way to access individual lines of a text file uses the `split` function to return an array of text lines. For example:

```
mytext = ""
readtextfile( "C:/test.txt", mytext )
lines = split( mytext, "\n" )
outln("There are " + length(lines) + " lines of text in the file.")
if (length(lines) > 5) outln("Line 5: '", lines[5], "'")
```

6.2 File System Functions

Suppose you want to run SAM with many different weather files, and consequently need a list of all the files in a folder that have the *.tm2* extension. SolScript provides the `directorylist` function to help out in this situation. If you want to filter for multiple file extensions, separate them with commas.

```
file_names = directorylist( "C:/Windows", "dll" ) ' could also use "txt,dll"
outln("Found " + length(file_names) + " files that match.")
outln(unsplit(file_names, "\n"))
```

To list all the files in the given folder, leave the extension string empty or pass in `"*"`.

Sometimes you need to be able to quickly extract the file name from the full path, or vice versa. The functions `filenameonly` and `dirnameonly` extract the respective sections of the file name, returning the result.

To test whether a file or directory exist, use the `direxists` or `fileexists` functions. Examples:

```
path = "C:/SAM/2009.8.13/samsim.dll"
dir = dirnameonly( path )
name = filenameonly( path )
outln( "Path: " + path )
outln( "Name: " + name + " Exists? " + fileexists(path) )
outln( "Dir: " + dir + " Exists? " + direxists(dir))
```

6.3 Standard Dialogs

To facilitate writing more interactive scripts, SolScript includes various dialog functions. We have already used the `notice` and `yesno` functions in previous examples.

The `choosefile` function pops up a file selection dialog to the user, prompting them to select a file. `choosefile` will accept three optional parameters: the path of the initial directory to show in the dialog, a wildcard filter like `"*.txt"` to limit the types of files shown in the list, and a dialog caption to display on the window. Example:

```
file = choosefile("c:/SAM", "*.dll", "Choose a DLL file")
if (file == "")
   notice("You did not choose a file, quitting.")
   exit
else
   if ( not yesno("Do you want to load:\n\n" + file)) exit

   ' proceed to load .dll file
   outln("Loading " + file)
end
```

6.4 Calling Other Programs

Suppose you have a program on your computer that reads an input file, makes some complicated calculations, and writes an output file.

There are two very similar ways to call external programs: the `system` and `shell` functions. They are identical except that `shell` pops up an interactive system command window and runs the program in it. Both functions will wait until the called program finishes before returning to SolScript, so that the program runs *synchronously*. Examples:

```
system("notepad.exe") ' run notepad and wait
shell("ipconfig /all > c:/test.txt")
output = ""
readtextfile("c:/test.txt", output)
outln(output)
```

Each program runs in a folder that the program refers to as the *working directory*. Sometimes you may need to switch the working directory to conveniently access other files, or to allow an external program to run correctly.

```
working_dir = cwd()     ' get the current working directory
chdir( "C:/windows" )   ' change the working directory
outln("cwd=" + cwd() )
chdir(working_dir)      ' change it back to the original one
outln("cwd=" + cwd() )
```

Created with the Personal Edition of HelpNDoc: Generate EPub eBooks with ease

Function Libraries

The functions described in this section comprise the SolScript standard library of function calls for data manipulation, operating system access, and calculation.

Created with the Personal Edition of HelpNDoc: Free PDF documentation generator

Types and Data

Type/Data Manipulation

TypeOf (<VARIANT>):STRING
Returns a description of the argument type.

Integer (VARIANT):INTEGER
Converts the variable to an integer number.

Double (VARIANT):DOUBLE
Converts the variable to a double-precision floating point number.

Boolean (VARIANT):BOOLEAN
Converts the variable to a boolean.

String (...):STRING
Converts the given variables to a string.

IntegerArray (STRING):ARRAY
Converts a string delimited by {;, \t\n} to an integer array.

DoubleArray (STRING):ARRAY
Converts a string delimited by {;, \t\n} to a double-precision floating point array.

Length (ARRAY):INTEGER
Return the length of an array.

Array (...):ARRAY
Creates an array out of the argument list.

Allocate (INTEGER:PRIMARY, [INTEGER:SECONDARY]):ARRAY
Creates an empty array with the specified dimensions.

Resize (<ARRAY>, INTEGER:PRIMARY, [INTEGER:SECONDARY]):NONE
Resizes an array or 2D matrix.

Append (<ARRAY>, ...):NONE
Appends one or more items to an array.

Prepend (<ARRAY>, ...):NONE
Prepends one or more items to an array.

Created with the Personal Edition of HelpNDoc: Create HTML Help, DOC, PDF and print manuals from 1 single source

Input and Output

Input/Output

Out (...):NONE
Print data to the output device.

OutLn (...):NONE
Print data to the output device followed by a newline.

Print (STRING:Format, ...):NONE
Print formatted data to the output device using an extended 'printf' syntax.

In (...):STRING
Request input from the input device, showing an optional prompt.

Notice (...):NONE
Show a message dialog.

YesNo (...):BOOLEAN)
Show a Yes/No dialog. Returns true if yes was clicked

ChooseFile ([STRING:Initial dir], [STRING:Filter], [STRING:Caption]):STRING
Show a file selection dialog, with optional parameters.

Open (STRING:File, STRING:Mode):INTEGER
Opens a file for reading 'r', writing 'w', or appending 'a'.

Close (INTEGER:FileNum):NONE
Closes a file.

Seek (INTEGER:FileNum, INTEGER:Offset, INTEGER:Origin):INTEGER
Sets the position in an open file.

Tell (INTEGER:FileNum):INTEGER
Returns the current file position.

Eof (INTEGER:FileNum):BOOLEAN
Determines whether a file is at the end.

Write (INTEGER:FileNum, ...):BOOLEAN
Writes data as text to a file.

WriteN (INTEGER:FileNum, VARIANT data, INTEGER: NumChars):BOOLEAN
Writes character data to a file.

WriteLn (INTEGER:FileNum, VARIANT data):BOOLEAN
Writes a line to a file as a string.

ReadN (INTEGER:FileNum, <STRING>:Data, INTEGER:NumChars):BOOLEAN
Reads characters from a file.

ReadLn (INTEGER:FileNum, <STRING>:Line):BOOLEAN
Reads a line from a file, returning false if no more lines exist.

ReadFmt (INTEGER:filenum, STRING:format=[idgefxsb]*, STRING:delimiters, ... VALUE ARGUMENT LIST):BOOLEAN
Reads a data line from a file with the given sequence of types and delimiters. Number of value arguments must equal number of characters in format string

OpenWF (STRING:file, [ARRAY:header info]):INTEGER
Opens a weather (TM2, TM3, EPW) file for reading.

ReadWF (INTEGER:filenum, ARRAY:y|m|d|h|gh|dn|df|wind|tdry|twet|relhum|pres *or* [INTEGER:y, INTEGER:m, INTEGER:d, INTEGER:h, DOUBLE:gh, DOUBLE:dn, DOUBLE:df, DOUBLE:wind, DOUBLE:tdry, DOUBLE:twet, DOUBLE:relhum, DOUBLE:pres]):BOOLEAN
Reads a line of data from a weather file.

CustomizeTMY3 (STRING:Source tmy3 file, STRING:Target tmy3 file, [STRING:Column name=gh|dn|df|tdry|twet|wind|pressure|relhum, ARRAY:Values(8760)]*):BOOLEAN
Overwrites columns of 8760 data in a TMY3 file and writes a new file.

WriteTextFile (STRING:Filename, VARIANT data):BOOLEAN
Writes a file of text data to disk. Returns true on success.

ReadTextFile (STRING:Filename, <STRING>:Data):BOOLEAN
Reads a text file from disk, returning true on success.

GetHomeDir (NONE):STRING
Returns the current user's home directory.

Cwd (NONE):STRING

Returns the current working directory.

ChDir (STRING: Path):BOOLEAN
Change the current working directory.

DirectoryList (STRING:Path, STRING:Comma-separated extensions, [BOOLEAN:Include folders]):ARRAY
Enumerates all the files in a directory that match a comma separated string of extensions.

System (STRING):INTEGER
Run a system command, returning the process exit code.

Shell (STRING):BOOLEAN
Run a system command in a new console window. Returns true on success.

FileNameOnly (STRING:Path):STRING
Returns only the file name portion of a full path.

DirNameOnly (STRING:Path):STRING
Returns only the directory portion of a full path.

Extension (STRING:File):STRING
Returns the extension of a file.

DirExists (STRING:Path):BOOLEAN
Returns true if the specified directory exists.

FileExists (STRING:Path):BOOLEAN
Returns true if the specified file exists.

CopyFile (STRING:File1, STRING:File2):BOOLEAN
Copies file 1 to file 2.

RenameFile (STRING:File1, STRING:File2):BOOLEAN
Renames file 1 to file 2.

DeleteFile (STRING:File):BOOLEAN
Deletes the specified file.

MkDir (STRING:Path):BOOLEAN
Creates a directory including the full path to it.

RmDir (STRING:Path):BOOLEAN
Deletes a directory and everything it contains.

Decompress (STRING:Archive, STRING:Target):BOOLEAN
Decompresses an archive file (ZIP, TAR, TAR.GZ, GZ).

HttpGet (STRING:Url):STRING
Performs an HTTP web query and returns the result as plain text.

HttpDownload (STRING:Url, STRING:LocalFile):BOOLEAN

Downloads a file form the web, showing a progress dialog.

Created with the Personal Edition of HelpNDoc: Easily create PDF Help documents

Math

Math

Mod (INTEGER, INTEGER):INTEGER
Returns the remainder after X is divided by Y

Abs (NUMBER):NUMBER
Absolute value of the number.

Min (NUMBER, NUMBER *or* ARRAY):NUMBER
Returns the smaller of two values, or the smallest in an array.

Max (NUMBER, NUMBER *or* ARRAY):NUMBER
Returns the larger of two values, or the largest in an array

Ceil (NUMBER):DOUBLE
Returns the number rounded up to the nearest integer.

Floor (NUMBER):DOUBLE
Returns the number rounded down to the nearest integer.

Sqrt (NUMBER):DOUBLE
Returns the square root of a number.

Pow (NUMBER:X, NUMBER:Y):DOUBLE
Returns 'X' raised to the 'Y' power.

Exp (NUMBER):DOUBLE
Returns the exponential value, base 'e'.

Log (NUMBER):DOUBLE
Returns the logarithm of a number, base 'e'.

Log10 (NUMBER):DOUBLE
Returns the logarithm of a number, base 10.

Sin (NUMBER):DOUBLE
Returns the sine of a radian value.

Cos (NUMBER):DOUBLE
Returns the cosine of a radian value.

Tan (NUMBER):DOUBLE
Returns the tangent of a radian value.

ASin (NUMBER):DOUBLE
Returns the arcsine of a number in radians.

ACos (NUMBER):DOUBLE
Returns the arccosine of a number in radians.

ATan (NUMBER):DOUBLE
Returns the arctangent of a number in radians.

ATan2 (NUMBER:Y, NUMBER:X):DOUBLE
Returns the arctangent of 'Y'/'X' in radians.

IsNan (DOUBLE):BOOLEAN
Returns true if the number is NAN.

NanVal (NONE):DOUBLE
Returns NAN.

UnifRand (NONE):DOUBLE
Returns a random number with uniform distribution (0..1).

NormRand (NONE):DOUBLE
Returns a random number with normal distribution around 0.

Created with the Personal Edition of HelpNDoc: iPhone web sites made easy

String Manipulation

String Manipulation

StrPos (STRING, STRING:Search):INTEGER
Returns the first position of the search string, or -1 if not found.

StrRPos (STRING, STRING:Search):INTEGER
Returns the first position of the search string from the right, or -1 if not found.

StrLeft (STRING, INTEGER:N):STRING
Returns the left 'N' character string.

StrRight (STRING, INTEGER:N):STRING
Returns the right 'N' character string.

StrLower (STRING):STRING
Returns a lower case version of the string.

StrUpper (STRING):STRING
Returns an upper case version of the string.

StrMid (STRING, INTEGER:Start, [INTEGER:Count]):STRING
Returns the substring from the specified start position, of length 'count'. If 'count' is not supplied, the remainder of the string is returned.

StrLen (STRING):INTEGER
Returns the length of a string.

StrReplace (STRING, STRING:s0, STRING:s1):STRING
Returns a string with all instances of 's0' replaced with 's1'.

StrCmp (STRING:s0, STRING:s1):INTEGER
Case-sensitive comparison. Returns 0 if equal, positive if s0 comes before s1, and negative if s1 comes before s0.

StrICmp (STRING:s0, STRING:s1):INTEGER
Case-insensitive comparison. Returns 0 if equal, positive if s0 comes before s1, and negative if s1 comes before s0.

StrGCh (STRING, INTEGER:position):STRING
Gets the character at the specified position.

StrSCh (STRING, INTEGER:position, STRING:char):NONE
Sets the character at the specified position.

Split (STRING, STRING:delimiters):ARRAY
Splits the string into an array.

Unsplit (ARRAY, STRING:delimiters):STRING
Unsplits an array into a string.

Format (STRING:Format, ...):STRING
Formats data into a string using an extended 'printf' syntax.

Created with the Personal Edition of HelpNDoc: Free help authoring tool

Matrices

Matrix Functions

euler3 (ARRAY:Origin, ARRAY:Aimpoint, DOUBLE:Z rotation):ARRAY
Calculates 3D Euler angles.

reftoloc (ARRAY:euler):MATRIX
Returns 3x3 transform matrix from euler angles from ref to loc.

loctoref (ARRAY:euler):MATRIX
Returns 3x3 transform matrix from euler angles from loc to ref.

tr2local (ARRAY:Posref, ARRAY:Cosref, ARRAY:Origin, MATRIX:Rreftoloc,

`<ARRAY:Posloc>, <ARRAY:Cosloc>):NONE`
Transforms a point from a reference frame to a local frame.

tr2reference `(ARRAY:Posloc, ARRAY:Cosloc, ARRAY:Origin, MATRIX:Rloctoref, <ARRAY:Posref>, <ARRAY:Cosref>):NONE`
Transforms a point from a local frame to a reference frame.

v3 `( [DOUBLE:x, DOUBLE:y, DOUBLE:z] ):ARRAY`
Returns a 3 element vector, optionally with values.

dot3 `(ARRAY:a, ARRAY:b):DOUBLE`
Returns the dot product of two 3 element vectors.

transpose3x3 `( MATRIX ): MATRIX`
Returns the transpose of a 3x3 matrix.

mxv3 `( MATRIX:M, ARRAY:V ):ARRAY`
Returns matrix M multiplied by vector V in 3 dimensions.

Created with the Personal Edition of HelpNDoc: Free help authoring tool

SolTrace Library

SolTrace Functions

st_filename `( NONE ):STRING`
Returns the current SolTrace project file name.

st_screen `( INTEGER:Screen number ):NONE`
Switch the currently visible SolTrace screen.

st_save `( [STRING:File name] ):BOOLEAN`
Saves the current SolTrace project to disk. A file name can be optionally specified.

st_close `( NONE ):NONE`
Closes the current SolTrace project without checking if it was modified.

st_open `( STRING:File name ):BOOLEAN`
Open a SolTrace project file from the disk.

st_num_stages `( NONE ):INTEGER`
Returns the number of stages in the system geometry

st_add_stage `( STRING:Stage name ):INTEGER`
Add a stage to the system geometry

st_delete_stage `( STRING:Stage name ):NONE`
Deletes a stage from the system geometry

st_clear_stages (NONE):NONE
Deletes all stages from the system geometry

st_stage_flags (INTEGER:stage num, [ARRAY:virtual|multihit|tracethrough *or* BOOLEAN:virtual, BOOLEAN:multihit, BOOLEAN:tracethrough]):[ARRAY]
Sets stage flags.

st_stage_name (INTEGER:stage num, [STRING:name]):[STRING]
Gets or sets the stage name.

st_stage_xyz (INTEGER:stage num, [ARRAY:xyz *or* DOUBLE:x,DOUBLE:y,DOUBLE:z]):[ARRAY]
Gets or sets the stage origin X Y Z.

st_stage_aim (INTEGER:stage num, [ARRAY:axayaz *or* DOUBLE:ax,DOUBLE:ay,DOUBLE:az]):[ARRAY]
Gets or sets the stage aim point AX AY AZ

st_stage_zrot (INTEGER:stage num, [DOUBLE:zrot]):[DOUBLE]
Gets or sets the stage Z rotation.

st_sun_pointsource ([BOOLEAN:pt src]):[BOOLEAN]
Gets or sets whether the sun is a point-source.

st_sun_shape ([STRING:shape]):[STRING]
Gets or sets the sun shape: Gaussian, Pillbox, User defined.

st_sun_sigma ([DOUBLE:Sigma]):[DOUBLE]
Gets or sets the sun sigma parameter.

st_sun_halfwidth ([DOUBLE:Halfwidth]):[DOUBLE]
Gets or sets the sun halfwidth parameter.

st_sun_xyz ([ARRAY:xyz *or* DOUBLE:x, DOUBLE:y, DOUBLE:z]):[ARRAY]
Gets or sets the X Y Z location of the sun.

st_sun_ldh ([ARRAY:ldh *or* DOUBLE:l, DOUBLE:d, DOUBLE:h]):[ARRAY]
Gets or sets the Latitude,Day,Hour specification of the sun position.

st_ldh_to_xyz (ARRAY:ldh):ARRAY
Converts the Latitude,Day,Hour representation to X,Y,Z coordinates.

st_sun_setuserdata (ARRAY:x, ARRAY:y):NONE
Sets the sun shape data points.

st_sun_getuserdata (<ARRAY:x>, <ARRAY:y>):NONE
Gets the sun shape data points.

st_num_optics (NONE):INTEGER
Returns the current number of optical property sets.

st_add_optic ([STRING:name]):NONE
Adds a new optical property set, optionally with a name.

st_delete_optic (INTEGER:index):NONE
Deletes the specified optical property set.

st_clear_optics (NONE):NONE
Deletes all optical property sets.

st_optic_name (INTEGER:idx [, STRING:name]):[STRING]
Gets or sets the name of the specified optical property set.

st_optic_property (INTEGER:idx, STRING:'front' or 'back', STRING:property name(s), VARIANT LIST:property values...):ARRAY
Gets or sets a variety of optical properties.

st_optic_property_list (NONE):ARRAY
Returns a list of all valid optical property names.

st_active_stage ([INTEGER:stage num]):[INTEGER]
Gets or sets the current active stage in the system for using element manipulation functions.

st_num_elements (NONE):INTEGER
Returns the current number of elements.

st_add_elements (INTEGER):INTEGER
Adds the specified number of elements to the table.

st_delete_element (INTEGER):INTEGER
Deletes the element at the specified index.

st_clear_elements (NONE):NONE
Clears the element table.

st_element_xyz (INTEGER:index [,DOUBLE:x,DOUBLE:y,DOUBLE:z *or* ARRAY:xyz]):[ARRAY]
Gets or sets the X,Y,Z origin for the specified element.

st_element_aim (INTEGER:index [,DOUBLE:ax,DOUBLE:ay,DOUBLE:az *or* ARRAY:axayaz]):[ARRAY]
Gets or sets the AX,AY,AZ aim point for the specified element.

st_element_zrot (INTEGER:index [,DOUBLE:zrot]):[DOUBLE]
Gets or sets the Z rotation for an element.

st_element_aperture (INTEGER:index [, ARRAY:aperture data]):[ARRAY]
Gets or sets the aperture data for an element.

st_element_surface (INTEGER:index [, ARRAY:surface data]):[ARRAY]
Gets or sets the surface data for an element.

st_element_interaction (INTEGER:index [, STRING:Reflection or Refraction]):[STRING]
Gets or sets the interaction type for an element.

st_element_optic (INTEGER:index [, STRING:optic type name]):[STRING]
Gets or sets the optic type for an element.

st_element_comment (INTEGER:index [, STRING:name]):[STRING]
Gets or sets the comment for an element.

st_setup_trace ([ARRAY *or* INTEGER:nrays, INTEGER:nmaxrays, INTEGER:nmemblocks, INTEGER:ncpucores]):[ARRAY]
Gets or sets trace parameters.

st_trace (NONE):NONE
Run the ray trace algorithm.

st_export_csv (STRING:File name):BOOLEAN
Export all ray intersection data as a CSV file.

st_num_intersect (NONE):INTEGER
Returns the number of ray intersections.

st_ray_xyz (INTEGER:index [, <DOUBLE:x>, <DOUBLE:y>, <DOUBLE:z>]):BOOLEAN or ARRAY
Returns X,Y,Z location data for a given intersection point. Fills in x,y,z parameters or returns array.

st_ray_cos (INTEGER:index [, <DOUBLE:x>, <DOUBLE:y>, <DOUBLE:z>]):BOOLEAN or ARRAY
Returns X,Y,Z cosine angles for a given intersection point. Fills in cosine x,y,z parameters or returns array.

st_element_map (INTEGER:idx):INTEGER
Returns the element number for a given intersection point.

st_stage_map (INTEGER:idx):INTEGER
Returns the stage number for a given intersection point.

st_ray_num (INTEGER:idx):INTEGER
Returns the ray number for a given intersection point.

st_sun_ray_count (NONE):INTEGER
Returns the number of sun rays generated.

st_count_intersections (INTEGER:stage_num, INTEGER:element_num):INTEGER)
Returns the number of ray intersections for an element. (0-based index numbers)

st_sun_extent (NONE):INTEGER
Returns the calculated extent of the sunshape.

Created with the Personal Edition of HelpNDoc: Benefits of a Help Authoring Tool

References

Steele, C.R., Balch, C.D., Jorgensen G.J., Wendelin, T. and Lewandowski, A., 1991, “Membrane Dish Analysis: A Summary of Structural and Optical Analysis Capabilities,” NREL/TP-253-3432, National Renewable Energy Laboratory, Golden, CO.

Ratzel, A.C., and Boughton, B.D., 1987, “CIRCE.001: A Computer Code for Analysis of Point-Focus Concentrators with Flat Targets,” SAND86-1866, Sandia National Laboratories, Albuquerque, NM.

Spencer, G.H., and Murty, M.V.R.K., 1962, “General Ray-Tracing Procedure,” *Journal of the Optical Society of America*, Vol, 52, June, pp.672-678.

Neumann, A., A. Witzke, S.A. Jones, G. Schmitt, "Representative Terrestrial Solar Brightness Profiles," *Journal of Solar Energy Engineering*, 124, pp. 198-204, May, 2002.

Created with the Personal Edition of HelpNDoc: Free HTML Help documentation generator