Floating Point Benchmark

This is John Walker's Floating Point Benchmark, derived from…

Marinchip Interactive Lens Design System
John Walker December 1980

by John Walker
http://www.fourmilab.ch/

This is a complete optical design raytracing algorithm, stripped of its user interface and recast into *Mathematica*. It not only determines execution speed on an extremely floating point (including trig function) intensive real-world application, it checks accuracy on an algorithm that is exquisitely sensitive to errors. The performance of this program is typically far more sensitive to changes in the efficiency of the trigonometric library routines than the average floating point program.

Implemented in November 2016 by John Walker.

Spectral Lines

We define names for standard spectral lines in Angstroms. (Not all are used in this program.)

In[41]:=
```
aLine = 7621.0;
bLine = 6869.955;
cLine = 6562.816;
dLine = 5895.944;
eLine = 5269.557;
fLine = 4861.344;
gPrimeLine = 4340.477;
hLine = 3968.494;
```

Lens Design

The test case used in this program is the design for a 4 inch f/12 achromatic telescope objective used as the example in Wyld's classic work on ray tracing by hand, given in *Amateur Telescope Making*, Volume 3 (Volume 2 in the 1996 reprint edition).

Definitions for accessing fields of a surface description in a design.

In[49]:=
```
curvatureRadius = 1;
indexOfRefraction = 2;
dispersion = 3;
edgeThickness = 4;
```

In[53]:=
```
wyldClearAperture = 4;
wyldLens = {
              {27.05, 1.5137, 63.6, 0.52},
              {-16.68, 1.0, 0.0, 0.138},
              {-16.68, 1.6164, 36.7, 0.38},
              {-78.1, 1.0, 0.0, 0.0}
            };
```

traceContext

A traceContext is a list consisting of the following fields:

axialIncidence Is the ray paraxial or marginal?

radiusOfCurvature Radius of curvature of surface being crossed. If 0, surface is plane.

objectDistance Distance of object focus from lens vertex. If 0, incoming rays are parallel and the following must be specified:

rayHeight Height of ray from axis. Only relevant if objectDistance == 0

axisSlopeAngle Angle incoming ray makes with axis at intercept

fromIndex Refractive index of medium being left

toIndex Refractive index of medium being entered

Definitions for accessing fields in a traceContext list.

In[55]:=
```
axialIncidence = 1;
radiusOfCurvature = 2;
objectDistance = 3;
rayHeight = 4;
axisSlopeAngle = 5;
fromIndex = 6;
toIndex = 7;
```

transitSurface

transitSurface carries a ray across a surface. The function is called with the elements of the traceContext as its arguments. It returns a traceContext as a list with the following components modified to reflect the ray as it exits the surface.

objectDistance Distance from vertex to object focus after refraction

rayHeight Height of ray from axis

axisSlopeAngle Angle incoming ray makes with axis at intercept after refraction

There are four cases: a paraxial or marginal ray, each crossing a flat or curved surface.

First, a paraxial ray crossing a flat surface.

In[62]:=
```
transitSurface[paraxialRay, radiusOfCurvatureS_, objectDistanceS_,
  rayHeightS_, axisSlopeAngleS_, fromIndexS_, toIndexS_] :=
 {paraxialRay, 0, objectDistanceS * (toIndexS / fromIndexS),
   rayHeightS, axisSlopeAngleS * (fromIndexS / toIndexS),
   fromIndexS, toIndexS} /; radiusOfCurvatureS == 0
```

Second, a paraxial ray crossing a curved surface.

In[63]:=
```
transitSurface[paraxialRay, radiusOfCurvatureS_, objectDistanceS_,
  rayHeightS_, axisSlopeAngleS_, fromIndexS_, toIndexS_] :=
 Block[{odz, axisSlopeAngleP, iangSin, rangSin, axisSlopeAnglePP,
    rayHeightP, objectDistanceP}, odz = objectDistanceS == 0;
   axisSlopeAngleP = If[odz, 0, axisSlopeAngleS];
   iangSin = If[odz, rayHeightS / radiusOfCurvatureS,
     ((objectDistanceS - radiusOfCurvatureS) / radiusOfCurvatureS) *
      axisSlopeAngleS]; rangSin = (fromIndexS / toIndexS) * iangSin;
   axisSlopeAnglePP = axisSlopeAngleP + iangSin - rangSin;
   rayHeightP = If[odz, rayHeightS, objectDistanceS * axisSlopeAngleP];
   objectDistanceP = rayHeightP / axisSlopeAnglePP;
   {paraxialRay, radiusOfCurvatureS, objectDistanceP, rayHeightP,
    axisSlopeAnglePP, fromIndexS, toIndexS}] /; radiusOfCurvatureS ≠ 0
```

Third, a marginal ray crossing a flat surface.

In[64]:=
```
transitSurface[marginalRay, radiusOfCurvatureS_, objectDistanceS_,
  rayHeightS_, axisSlopeAngleS_, fromIndexS_, toIndexS_] :=
 With[{rang = -(ArcSin[(fromIndexS / toIndexS)]) * Sin[axisSlopeAngleS]},
   {marginalRay, 0, objectDistanceS *
     ((toIndexS * Cos[-rang]) / (fromIndexS * Cos[axisSlopeAngleS])),
    rayHeightS, -rang, fromIndexS, toIndexS}] /; radiusOfCurvatureS == 0
```

Fourth and finally, a marginal ray crossing a curved surface.

In[65]:=
```
transitSurface[marginalRay, radiusOfCurvatureS_, objectDistanceS_,
  rayHeightS_, axisSlopeAngleS_, fromIndexS_, toIndexS_] :=
 Block[{odz, axisSlopeAngleP, iangSin, iang, rangSin, axisSlopeAnglePP,
   sagitta, rayHeightP, objectDistanceP}, odz = objectDistanceS == 0;
  axisSlopeAngleP = If[odz, 0, axisSlopeAngleS];
  iangSin = If[odz, rayHeightS / radiusOfCurvatureS,
    ((objectDistanceS - radiusOfCurvatureS) / radiusOfCurvatureS) *
     Sin[axisSlopeAngleS]]; rangSin = (fromIndexS / toIndexS) * iangSin;
  iang = ArcSin[iangSin];
  axisSlopeAnglePP = axisSlopeAngleP + iang - ArcSin[rangSin];
  sagitta = 2 * radiusOfCurvatureS * Sin[(axisSlopeAngleP + iang) / 2] ^ 2;
  rayHeightP = If[odz, rayHeightS, objectDistanceS * axisSlopeAngleP];
  objectDistanceP = (radiusOfCurvatureS *
      Sin[axisSlopeAngleP + iang] * Cot[axisSlopeAnglePP]) + sagitta;
  {marginalRay, radiusOfCurvatureS, objectDistanceP, rayHeightP,
   axisSlopeAnglePP, fromIndexS, toIndexS}] /; radiusOfCurvatureS ≠ 0
```

traceLine

traceLine performs a ray trace for a given design for a specific spectral line and ray height. The caller passes in the design and desired spectral line, along with a traceContext initialised with the required axialIncidence, radiusOfCurvature (0), objectDistance (0), rayHeight, axisSlopeAngle (0), and fromIndex (1). We walk through the design list, applying transitSurface for each surface encountered, and return a traceContext whose objectDistance and axisSlopeAngle contain the results of the ray trace.

In[66]:=
```
traceLine[design_, spectralLine_, context_] := context /; Length[design] == 0
```

In[67]:=
```
traceLine[design_, spectralLine_, context_] :=
 Block[{surf, toIndexP, contextP}, surf = First[design];
  toIndexP = If[surf[[indexOfRefraction]] > 1,
    surf[[indexOfRefraction]] + ((dLine - spectralLine) / (cLine - fLine)) *
      ((surf[[indexOfRefraction]] - 1) / surf[[dispersion]]), surf[[
     indexOfRefraction]]]; contextP = transitSurface[context[[axialIncidence]],
    surf[[curvatureRadius]], context[[objectDistance]], context[[rayHeight]],
    context[[axisSlopeAngle]], context[[fromIndex]], toIndexP];
  (* Print["traceLine: ", surf, " ",context, " ",contextP]; *)
  traceLine[Rest[design], spectralLine,
   {contextP[[axialIncidence]], contextP[[radiusOfCurvature]],
    contextP[[objectDistance]] - surf[[edgeThickness]], contextP[[rayHeight]],
    contextP[[axisSlopeAngle]], contextP[[toIndex]], 0}]] /; Length[design] > 0
```

traceLens

The **traceLens** function is a little bit of syntactic sugar to simplify invoking **traceLine**. It creates an initial traceContext for **traceLine** which specifies the axialIncidence and rayHeight (computed from the clear aperture of the design), invokes **traceLine** with the

supplied design and spectral line, then returns a list containing the objectDistance and axisSlopeAngle resulting from the ray trace.

Definitions to access fields in the result from **traceLens**:

In[68]:=
```
tlOD = 1;
tlSA = 2;

traceLens[design_, clearAperture_, spectralLine_, axialIncidenceS_] :=
 With[{context = traceLine[design, spectralLine,
     {axialIncidenceS, 0, 0, clearAperture / 2, 0, 1, 0}]},
  {context[[objectDistance]], context[[axisSlopeAngle]]}]
```

evaluateDesign

The **evaluateDesign** function performs a ray trace on a given design with a specified clear aperture and returns a designEvaluation list which includes the results for the D line and calculation of spherical aberration, coma, and chromatic aberration, along with the conventional acceptable upper bounds for these quantities.

Definitions to access fields in the result from **evaluateDesign**:

In[71]:=
```
eDdMar = 1;
eDdPar = 2;
eDaberrLspher = 3;
eDaberrOsc = 4;
eDaberrLchrom = 5;
eDMaxLspher = 6;
eDMaxOsc = 7;
eDMaxAchrom = 8;

evaluateDesign[design_, clearApertureS_] :=
 With[{ dMar = traceLens[design, clearApertureS, dLine, marginalRay],
   dPar = traceLens[design, clearApertureS, dLine, paraxialRay],
   cMar = traceLens[design, clearApertureS, cLine, marginalRay],
   fMar = traceLens[design, clearApertureS, fLine, marginalRay]},
  (* Print["dM ", dMar, " dP ", dPar, " cM ", cMar, " fM ", fMar]; *)
  With[{aberrLspherS = dPar[[tlOD]] - dMar[[tlOD]], aberrOscS =
     1 - (dPar[[tlOD]] * dPar[[tlSA]]) / (Sin[dMar[[tlSA]]] * dMar[[tlOD]]),
    aberrLchromS = fMar[[tlOD]] - cMar[[tlOD]],
    maxLspherS = 0.0000926 / (Sin[dMar[[tlSA]]]^2)},
   (* Print["Lspher ", aberrLspherS, " OSC ", aberrOscS, " Lchrom ",
     aberrLchromS, " Max Aberr ", maxLspherS]; *){dMar, dPar,
    aberrLspherS, aberrOscS, aberrLchromS, maxLspherS, 0.0025, maxLspherS}]]
```

evaluationReport

evaluationReport edits the list result from **evaluateDesign** into primate-readable text

we can compare with the expected results. This function is not timed in the benchmark.

In[80]:=
```
nf[n_] := ToString[NumberForm[n // N, {14, 11}]]

evaluationReport[de_] :=
  "   Marginal ray          " <> nf[de[[eDdMar]][[tlOD]]] <> "    " <>
  nf[de[[eDdMar]][[tlSA]]] <> "\n" <> "   Paraxial ray          " <>
  nf[de[[eDdPar]][[tlOD]]] <> "    " <> nf[de[[eDdPar]][[tlSA]]] <> "\n" <>
  "Longitudinal spherical aberration:        " <> nf[de[[eDaberrLspher]]] <> "\n" <>
  "    (Maximum permissible):                " <> nf[de[[eDMaxLspher]]] <> "\n" <>
  "Offense against sine condition (coma):    " <> nf[de[[eDaberrOsc]]] <> "\n" <>
  "    (Maximum permissible):                " <> nf[de[[eDMaxOsc]]] <> "\n" <>
  "Axial chromatic aberration:               " <> nf[de[[eDaberrLchrom]]] <> "\n" <>
  "    (Maximum permissible):                " <> nf[de[[eDMaxAchrom]]] <> "\n"
```

validateResults

The **validateResults** function compares a primate-readable report from **evaluationReport** with the archival results from the reference implementation (which all language implementations must reproduce character-by-character [apart from trivia such as end of line conventions and trailing white space]). It returns a Boolean value indicating whether the results compared. This function is not timed in the benchmark.

In[82]:=
```
validateResults[er_] := With[{expectedResults =
    (* Reference results.These happen to be derived from a run
       on Microsoft Quick BASIC on the IBM PC/AT. *)
     "   Marginal ray          47.09479120920   0.04178472683\n" <>
     "   Paraxial ray          47.08372160249   0.04177864821\n" <>
     "Longitudinal spherical aberration:       -0.01106960671\n" <>
     "    (Maximum permissible):                0.05306749907\n" <>
     "Offense against sine condition (coma):    0.00008954761\n" <>
     "    (Maximum permissible):                0.00250000000\n" <>
     "Axial chromatic aberration:               0.00448229032\n" <>
     "    (Maximum permissible):                0.05306749907\n"},
  er == expectedResults]
```

Running the Benchmark

In[83]:=
```
standardNumberOfIterations = 1000;

runBenchmark[iters_] := Block[{ev, stime, etime, er}, stime = AbsoluteTime[];
  Do[ev = evaluateDesign[wyldLens, wyldClearAperture], {iters}];
  etime = AbsoluteTime[] - stime;
  er = evaluationReport[ev];
  If[! validateResults[er], Print["Error(s) in results!  This is VERY SERIOUS."];]
   Print["Execution time ", etime, " seconds for ", iters, " iterations."];
  Print["Execution time for ", standardNumberOfIterations,
   " iterations: ", etime * (standardNumberOfIterations / iters), " seconds."];]
```

Perform Benchmark

In[86]:=
```
runBenchmark[1000]
```

```
Execution time 5.609737 seconds for 1000 iterations.
Execution time for 1000 iterations: 5.609737 seconds.
```