Screen Space Approximate Gaussian Hulls

J. Meder\textsuperscript{1,2} and B. Brüderlin\textsuperscript{1}
\textsuperscript{1}TU Ilmenau, Germany
\textsuperscript{2}3DInteractive GmbH, Ilmenau, Germany

Abstract

The Screen Space Approximate Gaussian Hull method presented in this paper is based on an output sensitive, adaptive approach, which addresses the challenge of high quality rendering even for high resolution displays and large numbers of light sources or indirect lighting. Our approach uses dynamically sparse sampling of the light information on a low-resolution mesh approximated from screen space and applying these samples in a deferred shading stage to the full resolution image. This preserves geometric detail unlike common approaches using lower resolution rendering combined with upsampling strategies. The light samples are expressed by spherical Gaussian distribution functions, for which we found a more precise closed form integration compared to existing approaches. Thus, our method does not exhibit the quality degradation shown by previously proposed approaches and we show that the implementation is very efficient. Moreover, being an output sensitive approach, it can be used for massive scene rendering without additional cost.

CCS Concepts

- Computing methodologies → Rasterization; Reflectance modeling; Virtual reality; Image processing;

1. Introduction

High-quality lighting is still a significant challenge in interactive rasterization applications, even more so with screen resolutions moving to 4k and beyond. In VR applications this requirement is exacerbated further by high frame rates demanded. But even for consoles and mobile platforms resolution and frame rate demands are steadily growing. A common shortcut we have seen thus far is to carry out some or all per-pixel work on lower resolutions than the true target and using upscaling or splatting, sometimes combined with interpolation or extrapolation to give the impression of the higher target resolution. These approaches can however lose geometric detail which should be visible at the target resolution.

In this paper we present an experimental method and implementation using new means of approximation to allow resolution independence for hemispherical light gathering. Our method does not rely on precalculations or additional a priori information and purely works like a deferred shading approach on a G-Buffer. The basic idea is to approximate the screen space as a low-detail triangle mesh, only gathering light by means of mathematical distributions at the mesh’s vertex positions and integrate over these distributions after rasterization of this mesh. In detail, our contributions include

- a fast image segmentation approximation based on connected components
- a quick depth based tessellation of the screen space
- a novel closed form approximation convolving a spherical Gaussian with a cosine factor.

We first cover related work in section 2, while the rest of this paper is organized as follows: In section 3 we detail our screen space tessellation while section 4 concentrates on our work with and application of spherical Gaussians. Finally section 5 gives some empirical data and section 6 concludes this paper.

2. Related Work

Screen space lighting. The method we discuss in this paper does not shade in screen space directly, but uses geometric simplifications of the screen space as part of shading cost reduction. With similar ideas, Nichols et al. proposed a recursive subdivision of the screen space combined with splatting to reduce high rendering costs of many virtual point lights (VPLs) [NW09, NW10]. As inherent to splatting approaches, surface details get lost in the process, which they hide by manipulating the output in dependence of the surface normal. Ritschel et al. extend the popular screen space ambient occlusion method by lighting propagation to approximate global illumination in screen space [RGS09]. As an extension to the single layered conventional screen space, Nalbach et al. introduce the concept of deep screen space [NRS14] as a collection of micro-surfaces generated on-the-fly from visible geometry and splat calculated surfel lighting via an approach similar to aforementioned method of Nichols et al.

Tessellating the image space. Image space tessellation like the one we use and further explain in section 3 has mostly seen applications in depth based reprojection or Depth-Image-Based-Rendering
shading and interpolate them over the geometry, yielding plausible diffuse but only low-frequency specular lighting in fully dynamic scenes. Jendersie et al. also use such a cache based approach but assume a mostly static scene allowing the precomputation of radiosity values and surface interdependence regarding interreflections for these scene parts [JKG16]. This allows for much faster dynamic lighting of the static scene, but has severe limitations for dynamic objects. Our algorithm mainly fits into this last category of lighting approximation at the geometry level and, like Lensing et al., assumes a fully dynamic scene. Still, combinations with aforementioned complexity reductions at the light level are possible because we make no assumptions on the form of scene light representations.

3. Screen space approximate hulls

As outlined in section 1 we want to model the screen space as a low-poly mesh and only gather hemispherical lighting at this mesh’s vertices by distribution functions.

Among others, Lensing et al. [LB13] and Jendersie et al. [JKG16] already established the viability of sparsely computing lighting at discrete caches on the scene geometry with subsequent interpolation. While they placed these caches statically on the object level, we will dynamically place them using the currently visible screen space. We therefore do not need to precalculate and store these caches with the geometry and can use the complete cache budget in the visible space, allowing more detailed light capture. This is particularly important in the use case of massive scene rendering, where precomputations in object space may not be applicable due to time or storage space constraints. Also in contrast to the approach of Lensing et al., we do not use virtual lights as a representation of gathered light but rather mathematical approximations of the incident radiance \(L_i(\omega)\) (see equation (6) in section 4), more akin to the method of Jendersie et al. The main algorithm has three distinct steps:

1. Render a G-Buffer of the scene and downsample screen space depth to 1024x1024
2. Segment the downsampled depth into different regions of similar surfaces separated by depth edges.
3. For each segment of the downsampled depth draw an x-y bounding rectangle for light gathering and integration.

We will explain the reasoning behind these steps in the following sections in detail.

**Approximating the screen space mesh.** Our algorithm starts with a low-detail mesh of 32x32 quads of which the vertices are placed regularly in x-y direction of the screen space. For lighting its vertices are assigned a world space position read from the G-Buffer at the corresponding x-y position. As Figure 2b shows this does not suffice to create plausible lighting. Positions inside the triangles and G-Buffer pixel positions can differ substantially in depth edge regions. This is a similar problem Liktor et al. report [LD10]. Their solution however to simply exclude these triangles from rendering is not applicable here, as we want to cover every G-Buffer pixel with a fitting triangle. We therefore opt for mesh subdivision.

We experimented with different subdivision and triangulation strategies. Using the hardware tessellation based depth subdivision of Meder et al. [MB16] yielded good performance in terms of the tessellation itself but unfortunately creates too many vertices in depth edge regions which in turn induce a large overhead for the light gathering. Tying a Delaunay triangulation over depth edge points on the GPU remedied this problem but unfortunately was too slow on our target hardware. We also investigated the approaches of Müller et al. [MSD07] and Didyk et al. [DRE*10] but finally came to the conclusion that we actually don’t need the exact triangulations these methods provide: only a mesh covering each G-Buffer pixel, with the respective triangles residing closely to the pixels’ surfaces, is needed. The individual mesh parts may even overlap as long as we can choose the correct surface for each G-Buffer pixel in the end.

**Local depth segmentation.** With this knowledge, we do not really subdivide the individual 32x32 quads. Instead, we use a compute shader to apply local image segmentation to each quad’s corresponding depth region of the downsampled depth and create a new quad for each identified segment (see figure 2c). Corresponding to the local pixel count we use 32x32 as the work group size. As a base for the segmentation we use the connected component labeling proposed by He et al. [HCS08]. Connected components are identified using a simple depth difference compared to an adaptive threshold (comparison in function "swapLabel" in algorithm 1). The union find structure necessary for label equivalency resolves presented us with problems concerning thread contention and control flow which is why we currently omit this step. We instead propagate the component labels as given with function "segment" in algorithm 1 in right-down-left-up direction over the 32x32 pixel region each using 32 threads running line-wise or column-wise without contention. Of course, this heuristic potentially creates more segments than the original algorithm, which apart from generating additional light gathering overhead does not pose any problems, and we found it to work well in practice.

**Algorithm 1** Local region segmentation of one thread

```plaintext
function SWAPLABEL(x,y,dc,l,t,depth,label)
    d0 ← depth(x,y)
    if |dc − d0| > t(1 − dc) then
        l ← label(x,y)
    else
        label(x,y) ← l
    end if
    dc ← d0
end function

function PROPAGETE(x0,y0,xe,ye,t,depth,label)
    dc ← depth(x0,y0)
    l ← label(x0,y0)
    for x′ = x0,...,xe, y′ = y0,...,ye do
        swapLabel(x′,y′,dc,l,t,depth,label)
    end for
end function

function SEGMENT(x,y,t,depth,label)
    propagate(0,y,31,y,t,depth,label)
    propagate(x,0,31,t,depth,label)
    propagate(31,y,0,t,depth,label)
    propagate(x,31,0,t,depth,label)
end function
```
Finally, the screen space axis aligned bounding box of each identified segment is calculated: Each of the 32x32 threads takes the final label of its corresponding local pixel as an index in the local bounding box list and applies minimum/maximum atomic operations between its \((x, y, z)\) screen space coordinates and this bounding box’s minimum/maximum corners. Due to a limitation of atomic operations the coordinates have to be converted to integer, so we use integer image coordinates for \(x\) and \(y\) while for the pixel depth we simply use a direct bit conversion. Comparisons will still work under the IEEE 754 floating point definition, assuming all depth values being positive. The resulting bounding boxes are written back to GPU memory for rendering in the final lighting pass. Figure 2d shows a result of this approach.

**Light gathering, interpolation and integration.** The created mesh is subsequently rendered into the full viewport using the standard rasterization pipeline. Light gathering occurs in the vertex stage, where we accumulate the contributions of all present lights into a spherical Gaussian distribution to approximate the incident radiance \(L_i(x)\) (see section 4). How the individual light radiance values and directions are obtained depends on the type of light. For the polygonal area lights we use in section 5 we use the radiant exitance of the polygon times its projected area on the unit sphere around the vertex’s world position and accumulate all normalized directions from the world position to the polygon’s vertices to simulate the directional distribution in the spherical Gaussian.

Additionally in the vertex stage, we fetch the actual world space position of a quad vertex from the G-Buffer at the vertex’s \(x\)-\(y\) position, project it to screen space and compare the result to the quad’s screen space bounding box. If it lies outside, meaning the vertex overlaps into another depth region, we correct the world position: we take the intersections of the camera’s view direction with the quad’s segment bounding box planes and use their mean reprojected to world space.

Perspective-correct interpolation of the distributions’ parameters automatically occurs during rasterization of the mesh yielding a distribution per quad fragment. Green et al. already showed such an interpolation of the distribution’s parameters over object geometry to be sensible [GKMD06]. Final lighting is calculated by integrating the distribution (see section 4) with respect to the original viewport pixel’s surface hemisphere and parameters in the G-Buffer. To avoid applying fragments residing on a quad overlapping from another depth region we again use a test of the G-Buffer pixel’s depth against the region bounds of the current fragment, discarding all fragments failing the test.

### 4. Gaussian lighting integration

**Spherical Gaussians.** Generally, spherical Gaussians are defined for all normalized directions \(\omega\) in the whole sphere \(\Omega\) as

\[
G(\mu, \lambda, \phi, \omega) = \mu \lambda \phi(\omega^\top \omega - 1),
\]

with \(\mu\) being its mean coefficient, \(\lambda\) its sharpness and \(\phi\) its main direction. We use \(\otimes\) as the dot product in this paper. This distribution has a simple analytical integral over \(\Omega\) [Tok16]:

\[
A(\mu, \lambda) = \int_{\Omega} G(\mu, \lambda, \phi, \omega) d\omega = \frac{2\pi}{\lambda} \left(1 - e^{-2\lambda}\right).
\]

In section 4 we rely on the multiplication of two spherical Gaussians \(G(\mu_1, \lambda_1, \phi_1, \omega)\) and \(G(\mu_2, \lambda_2, \phi_2, \omega)\) which yields a new spherical Gaussian \(G(\mu_3, \lambda_3, \phi_3, \omega)\) with [XCM'B14]

\[
\begin{align*}
\lambda_3 &= |\lambda_1 \phi_1 + \lambda_2 \phi_2| \\
\phi_3 &= \frac{\lambda_1 \phi_1 + \lambda_2 \phi_2}{\lambda_3} \\
\mu_3 &= \mu_1 \mu_2 |\lambda_1 - \lambda_2| - \lambda_1 \lambda_2.
\end{align*}
\]

Tokuyoshi proposed to use Toksvig’s filtering [Tok05] to approximate a spherical Gaussian’s parameters [Tok15]. Due to the expensive operations needed we also use this method to accumulate a set of incoming light rays \(L\) with light directions \(\omega_t\) and color intensity values \(\mu_t\) at a screen space mesh’s vertices during hemispherical sampling (see section 3). Direction \(\phi\) and sharpness \(\lambda\) of the spherical Gaussian are estimated as follows:

\[
\begin{align*}
\phi' &= \frac{\sum_{i \in L} \mu_i \omega_{ti}}{\sum_{i \in L} \mu_i}, \\
\phi &= \frac{\phi'}{|\phi'|}, \\
\lambda &= \frac{|\phi'|}{1 - |\phi'|}.
\end{align*}
\]

We use separate distributions for each color channel so \(\mu_t\) denotes the respective channel value and followingly the corresponding mean coefficient \(\mu\) is calculated for each channel as

\[
\mu = \frac{\sum_{i \in L} \mu_i}{A(\lambda)},
\]

i.e. normalizing the spherical integral (2) by \(A(\lambda) = \frac{2\pi}{\lambda} (1 - e^{-2\lambda})\).

**The rendering equation.** When the approximate mesh of section 3 is rasterized, the spherical Gaussian distributions associated with its vertices will be automatically interpolated by the GPU to yield a single distribution per generated pixel. We now want to use these to calculate the final lighting of each pixel. In general, to calculate the reflected light of a surface point the rendering equation [Kaj86] has to be solved:

\[
L_o(x, \omega_o) = L_e(x, \omega_o) + \int_{\Omega} L_i(\omega) f_r(x, \omega, \omega_o)(\omega \otimes n) d\omega.
\]

Ignoring self-emittance \(L_e\) for the sake of simplicity, we denote the hemisphere above pixel surface point \(x\) in direction of the surface normal \(n\), i.e. where \((\omega \otimes n)\) is positive, as \(\Omega^+_n\). Thus far, we approximated \(L_i(\omega)\) which is given by the spherical Gaussian distribution. For the bidirectional reflectance distribution function (BRDF) \(f_r\), we assume, that we can approximate it as another spherical Gaussian, which enables usage of the well-defined multiplication. We can omit this step for the commonly used Lambert diffuse BRDF, as this BRDF is just a constant \(\frac{c}{\pi}\) being the surface’s albedo. For other BRDFs meaningful spherical Gaussian approximations have been proposed for Phong distributions and well-known microfacet BRDFs by Wang et al. [WRG'B09]. We will not focus any further on this topic as it exceeds the scope of this paper.

**Cosine convolution.** The remaining problem is to convolve the combined spherical Gaussian with the cosine factor \(\omega \otimes n\). We found two solutions for this in literature, one of which assumes the cosine
factor to be quasi-constant, which allows its extraction from the integral \[\int_{\Omega} G(\mu, \lambda, \phi, \omega) (\omega \circ n) d\omega \approx \max(0, \phi \circ n) \int_{\Omega} G(\mu, \lambda, \phi, \omega) d\omega \] (7)

By its underlying assumption, this solution only works for spherical Gaussians of sufficient sharpness. In other words, the spherical Gaussian is interpreted as a directional light with direction \(\phi\) and flux \(A(\mu, \lambda)\). A failure case we quickly encountered was the representation of broad area lights which yields noticeably wrong shading when using equation (7) as figure 3 shows.

![Figure 3](image)

Figure 3: (a) Assuming the cosine factor of the rendering integral to be constant leads to noticeably wrong shading compared to (b) the ground truth when this assumption is violated.

The other solution we encountered is to approximate the cosine factor itself as a spherical Gaussian, which allows to use both the spherical Gaussian multiplication (3) and analytical integral (2):

\[
\begin{align*}
\int_{\Omega} G(\mu, \lambda, \phi, \omega) (\omega \circ n) d\omega & \approx \int_{\Omega} G(\mu, \lambda, \phi, \omega) (1, 2, n, \omega) d\omega \\
& = \int_{\Omega} G(\mu', \lambda', \phi', \omega) d\omega = A(\mu', \lambda')
\end{align*}
\]

Tokuyoshi uses this solution to calculate diffuse lighting under a Lambert BRDF [Tok16]. However, it is impossible to completely reflect the distribution of a clamped cosine factor using approximation (8). As Figure 4 illustrates, trade-offs have to be made leading to overly bright or dark shading depending on the surface normal. Because of these shortcomings, we investigated new ways of calculating the convolution directly.

**Direct integration.** One observation when trying to solve equation (6) is the necessity to integrate in \(\Omega^+\) rather than \(\Omega\). The previous solutions circumvented this by either clamping the cosine factor or using a cosine-like distribution which always yields positive values. Approaching the cosine convolution directly, we need a formalization for the hemispherical convolution of a spherical Gaussian with a cosine in the cosine’s domain of positive values.

We found that two specific hemispherical integrals have easy analytical solutions when integrating in spherical coordinates: the integral in the upper hemisphere, i.e. when \(n\) equals \(\phi\) is given by

\[
A_u(\lambda) = \int_0^{2\pi} \int_0^\pi G(\lambda, n, \omega) \cos(\theta) \sin(\theta) d\theta d\phi
= \frac{2\pi}{\lambda^2} (e^{-\lambda} - 1 + \lambda),
\]

and the lower hemispherical integral when \(\phi\) equals -\(n\)

\[
A_b(\lambda) = \int_0^{2\pi} \int_0^\pi G(\lambda, -n, \omega) \cos(\theta) \sin(\theta) d\theta d\phi
= \frac{2\pi}{\lambda^2} (e^{-\lambda} - 1 - \lambda).
\]

where \(\omega = (\sin(\theta) \cos(\phi), \sin(\theta) \sin(\phi), \cos(\theta))^T\). Proof can be found in appendix A. We leave out \(\mu\) as it is a constant which can be pulled out of the integral. We express the general form of these integrals, when the angle \(\beta = \arccos(\mu \circ n)\) is between 0 and \(\pi\), by rotating the input direction \(\omega\) away from the original upper hemisphere by this angle. As spherical Gaussians are isotropic we can w.l.o.g. assume a rotation around the local x-axis and thus use a standard rotation matrix

\[
M(\beta) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\beta) & \sin(\beta) \\ 0 & -\sin(\beta) & \cos(\beta) \end{pmatrix}.
\]

Assuming further w.l.o.g. that locally \(\phi\) is the z-axis \((0, 0, 1)^T\) leaves us with the integral

\[
\begin{align*}
A_b(\lambda, \beta) = & \int_0^{2\pi} \int_0^\pi G(\lambda, (0, 0, 1)^T, M\omega) \cos(\theta) \sin(\theta) d\theta d\phi \\
= & \int_0^{2\pi} \int_0^\pi \lambda (\cos(\beta) \cos(\theta) - \sin(\beta) \sin(\theta) - 1) \cos(\theta) \sin(\theta) d\theta d\phi.
\end{align*}
\]
Unfortunately, we did not find an analytical solution for $A_h$ and previous work suggests that indeed none exists \cite{XMR11}. We therefore investigated the integral numerically.

![Figure 5: (a) Normalized curves of $A_h$ for $\lambda \in [10^{-5};700]$ progressively change from $0.5 \cos(\beta) + 0.5$ to $\max(0, \cos(\beta))$. (b) Curves $s(\lambda, \beta)$ directly calculated from numerically integrated $A_h$.](image)

Inspecting the absolute values depending on $\beta$ for different sharpnesses yielded no valuable hints for us at first, but a quite different picture presented itself when normalizing the curves to an image of $[0;1]$. Observing the curve shapes in figure 5a, the function becomes equivalent to the normalized cosine

$$\hat{\cos}(\beta) = 0.5 \cos(\beta) + 0.5$$

(13)

when $\lambda$ approaches zero and progressively turns into the clamped cosine

$$\langle \cos(\beta) \rangle = \max(0, \cos(\beta))$$

(14)

for $\lambda$ approaching infinity. Given the known analytical values $A_h(\lambda, 0) = A_u(\lambda)$ and $A_h(\lambda, \pi) = A_b(\lambda)$ we approximate $A_h$ generally using a basic interpolation of the form

$$A_b(\lambda, \beta) \approx c(\lambda, \beta) A_u(\lambda) + (1 - c(\lambda, \beta)) A_b(\lambda)$$

(15)

with $c$ being the appropriate normalized curve dependent on $\lambda$ as seen in figure 5a. For a closed form of $c$, from our observations we can again use an interpolation, this time of the known cosine curves

$$c(\lambda, \beta) \approx s(\lambda, \beta) \hat{\cos}(\beta) + (1 - s(\lambda, \beta)) \langle \cos(\beta) \rangle.$$  

(16)

Because we actually know the precise curves we can solve (16) for $s(\lambda, \beta)$ using the numerical ground truth of $c$. Some examples are given in figure 5b.

Now we need a closed form solution for $s$. We experimented with a range of distributions and came to the conclusion that $s$ could be related to $|\cos(\beta)|$. Additionally, Wang et al. suggested a close relation between the general hemispherical integral of a spherical Gaussian and sigmoid-shaped functions \cite{WRG09}. We therefore tried a logistic curve \cite{GQ38} of the form

$$s(\lambda, \beta) = \frac{u(\lambda)}{e^{r(\cos(\beta))} + 1}.$$  

(17)

Using non-linear automatic curve fitting via Matlab this function allowed good fits to the numerical ground truths for $s$, with root-mean-square errors (RMSEs) ranging from $4.38 \cdot 10^{-11}$ to $3.8 \cdot 10^{-3}$. As for the remaining values of $u$ and $t$, we first noticed we can use $\lambda \approx 0.2958 t + 0.5033$. Secondly, the values of $t$ in figure 6 imply some variant of a square root over $\lambda$, and we ultimately used

$$t(\lambda) = 2.1007 \sqrt{\frac{\lambda}{\lambda + 4.4653}}.$$  

(18)

with the numerical constants obtained from non-linear curve-fitting via Matlab. With this, we can now closely approximate the hemispherical convolution of a cosine factor with a spherical Gaussian and indeed, compared to the other two methods we discussed in the previous section, our method shows significantly better error values (see figure 7). Also, our new approximation is not necessarily more expensive, as it does not need a spherical Gaussian multiplication, mainly uses linear interpolations and integrals $A_u$ and $A_b$, largely consisting of the same factors, can be optimized to avoid unnecessary exponentials and divisions.
In the latter case, all quad-fragments are discarded by the segment bounds test in the fragment stage. Furthermore, screen space depth inaccuracies can cause the vertex stage bounding test to fail, wrongfully applying position correction to non-overlapping vertices and in turn causing suboptimal shading. This case is visible in figure 8 with small/medium differences plus noticeable difference edges.

Applicability. Any source providing hemispherical lighting may be used to gather, such as global illumination focused approaches beside RSM like voxel cone tracing [CNS*11]. Accelerating direct sources like image based lighting or discrete sets of analytical lights is also possible. Even ray tracing or approximations like screen space reflections are imaginable for glossy lighting. Highly specular reflections pose a limitation of our approach. The screen space mesh would need a tessellation to the G-Buffer pixel level in the worst case. Depending on the respective application’s accuracy requirements, more individual distributions are needed in general.

6. Conclusion and future work

In this paper, we have investigated a new variant of resolution decoupled deferred shading retaining the original resolution and have found a higher quality use of spherical Gaussians. The technique is well-suited for alleviating traditionally pixel-bound operations without a priori knowledge of the scene and is applicable both to direct and indirect lighting methods. Its output-sensitivity and non-reliance on precalculations make it particularly suitable for massive scene rendering. Our resulting experimental implementation shows promising results concerning speed and quality.

To solve the artifacts of our segmentation heuristic we plan to incorporate a more sophisticated depth downsampling, either preserving all local segments or giving depth bounds approximations, and to modify our segmentation heuristic to give a minimal local segmentation. Moving the algorithm entirely to world space and replacing the bounding box with a plane approximation promises better accuracy. Observing the improvements when using available RSM optimizations is also interesting. We further like to evaluate our method with other techniques for light gathering.

Finding precise closed form solutions or approximations using other BRDFs or replacing our distributions with anisotropic spherical Gaussians [XSD*13] present additional research opportunities.

Acknowledgments


Appendix A: Hemispherical convolution

We start with expression (9) given in section 4 and can immediately solve the outer integral and expand G:

$$A_u(\lambda) = 2\pi \int_0^{\frac{\pi}{2}} e^{\lambda \cos(\theta)} \cos(\theta) \sin(\theta) d\theta$$

(19)
Substituting \( t = \cos(\theta) \) yields
\[
A_u(\lambda) = 2\pi \int_0^{\frac{\pi}{2}} e^{\lambda(t-1)} dt.
\]
(20)

Pulling out \( \frac{1}{\lambda^2} \) and expanding the sum by \( e^{\lambda(t-1)} \lambda \) gives
\[
A_u(\lambda) = \frac{2\pi}{\lambda^2} \left[ \int_0^{\frac{\pi}{2}} e^{\lambda(t-1)} \lambda dt - \int_0^{\frac{\pi}{2}} e^{\lambda(t-1)} \lambda^2 t + e^{\lambda(t-1)} \lambda^2 dt \right],
\]
(21)
of which we can integrate the first integral directly, the second via
the inverse product rule and finally resubstitute \( \cos(\theta) \):
\[
A_u(\lambda) = \frac{2\pi}{\lambda^2} \left[ e^{\lambda(t-1)} - e^{\lambda(t-1)} \lambda \right]_{t=0}^{\frac{\pi}{2}}
\]
\[
= \frac{2\pi}{\lambda^2} \left[ e^{\lambda(\cos(\theta)-1)} (1 - \lambda \cos(\theta)) \right]_{t=0}^{\frac{\pi}{2}}.
\]
(22)

Thus, the integral value resolves to
\[
A_u(\lambda) = \frac{2\pi}{\lambda^2} (e^{-\lambda} - 1 + \lambda)
\]
(23)

\( A_b \) from equation (10) can be integrated analogously.

References


(C) 2018 The Author(s)


