Circular Integration Region

Version 5.5

R. Steven Turley

November 12, 2018

Contents

1. Introduction

1.1. History

You've probably guessed that, since this is version 5.4, there were earlier version of this document. A previous version, 4.2, was written to provide a justification of Chelsea Thangavelu's work in the summer of 2017. Version 5.0 was an expansion with examples added for specific unit testing of parts of the formula and implementations in python. It is targeted towards providing the theoretical and testing framework for Michael Greenburg's senior thesis. I've also simplified the previous derivations in Section 5.3 for transformation of the integration region into right triangles. Version 5.3 further expanded this work with examples for Julia code and work on incorporating the quadrature results into a Nyström technique for solving the rough mirror reflectance integral equation. Version 5.4 cleaned up some errors in the coordinate transformation and simplified the explanations. Version 5.5 addressed the fact that some of the coordinate transformations of previous

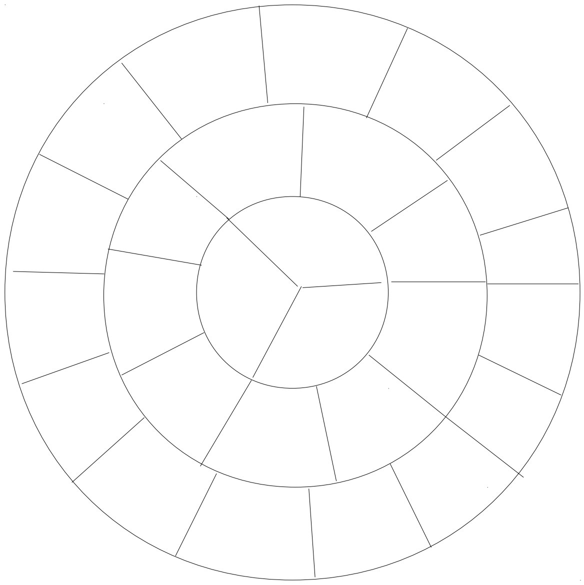

Figure 1: Annular patches with same area as center circle

versions involved improper rotations. I also updated some sign errors in the B Unit Tests (Section 5.5.4).

1.2. Justification

For a rough mirror in 3d, we will need to integrate over a 2d surface. Given the symmetries in our problem, it seemed advisable to make the integration region circular rather than rectangular. This eliminates sharp points on the surface and makes all points on the edge equidistant from the center. Dividing a circular region into patches is advisable in order to have basis functions with compact support[1] and compact regions to handle the integrable singularities in the integrand. Rectangular or triangular patches have well-developed high-order quadrature rules for integration, but can't produce a conformal circular boundary. Patches with equally spaced values of r and θ have the disadvantage of significantly different areas near the center of the surface and near the edges of the surface. A good compromise would be a central patch which is a circle divided into thirds and then annular patches with widths equal to the radius of the central circle divided into angular sections having the same area as the central circle regions as shown in Figure 1. If a is the radius of the central circle, the n^{th} annular ring has an area

$$A = \pi a^2(n+1)^2 - \pi a^2 n^2 \tag{1}$$

$$= \pi a^2(2n+1) \; . \tag{2}$$

[1]The Nystrom method doesn't use explicit basis functions, but they are implicit in the quadrature rules. Since Gaussian-type rules are exact for polynomials up to a certain order, those polynomials can be thought of as basis functions used in the expansion of the integrand.

It needs to be divided into $3(2n+1)$ patches for it to have the same area as the center circle arc $\pi a^2/3$.

2. Quadrature Rules

Abramowitz and Stegun have a four-point rule for integrating square regions that avoids the end points of integration[1]. The square rule is for integrating a region with

$$-h \leq x, y \leq h \ . \tag{3}$$

The rule approximates the integrals by evaluating the function at four points

$$x = \pm \frac{h}{\sqrt{3}} \tag{4}$$

$$y = \pm \frac{h}{\sqrt{3}} \ , \tag{5}$$

adding them together and multiplying the result by h^2. The rule has an error of order h^4. The square rule can be mapped onto a semi-circular annulus of inner radius an and angular range

$$\frac{2\pi m}{3(2n+1)} \leq \theta \leq \frac{2\pi(m+1)}{3(2n+1)} \tag{6}$$

$$0 \leq m \leq 6n+2 \ , \tag{7}$$

where m is the number of the semi-circular annulus. The worst case should be for a center circle region with $n=0$ and $m=0$. Mapping the square rule onto an annulus I have the following.

$$
\begin{aligned}
h &= \frac{a}{2} & (8)\\
r &= a\left[\left(n+\frac{1}{2}\right)+\frac{y}{2h}\right] & (9)\\
&= a\left(n+\frac{1}{2}\right)+y & (10)\\
&= a\left(n+\frac{1}{2}\pm\frac{1}{2\sqrt{3}}\right) & (11)\\
\theta &= 2\pi\frac{\left(m+\frac{1}{2}\right)+\frac{x}{2h}}{6n+3} & (12)\\
&= 2\pi\frac{a\left(m+\frac{1}{2}\right)+x}{3a(2n+1)} & (13)\\
&= 2\pi\frac{\left(m+\frac{1}{2}\right)\pm\frac{1}{2\sqrt{3}}}{3(2n+1)} & (14)\\
\frac{\partial r}{\partial y} &= 1 & (15)\\
\frac{\partial \theta}{\partial x} &= \frac{2\pi}{3a(2n+1)} & (16)
\end{aligned}
$$

$$
\begin{aligned}
\int_{an}^{a(n+1)} r\,dr\int_0^{2\pi/3} f(r,\theta)\,d\theta &\approx h^2\left(\frac{\partial r}{\partial y}\right)\left(\frac{\partial \theta}{\partial x}\right)\times\\
&\sum r\left(\pm\frac{h}{\sqrt{3}}\right) f\left(r\left(\pm\frac{h}{\sqrt{3}}\right),\theta\left(\pm\frac{h}{\sqrt{3}}\right)\right) & (17)\\
&\approx \left(\frac{a}{2}\right)^2\frac{2\pi}{3a(2n+1)}\sum\left[a\left(n+\frac{1}{2}\right)\pm\frac{a}{2\sqrt{3}}\right]f(r,\theta) & (18)\\
&\approx \frac{\pi a^2}{6(2n+1)}\sum\left[\left(n+\frac{1}{2}\pm\frac{1}{2\sqrt{3}}\right)\right]f(r(y),\theta(x)) & (19)
\end{aligned}
$$

Letting

$$
\begin{aligned}
r_\pm &= a\left(n+\frac{1}{2}\pm\frac{1}{2\sqrt{3}}\right) & (20)\\
\theta_\pm &= 2\pi\frac{\left(m+\frac{1}{2}\right)\pm\frac{1}{2\sqrt{3}}}{3(2n+1)}, & (21)
\end{aligned}
$$

this can be written for the entire circle as

$$
\begin{aligned}
\int_0^s r\,dr\int_0^{2\pi} f(r,\theta)\,d\theta \approx \frac{\pi a^2}{12}\sum_{n=0}^{N-1}\sum_{m=0}^{2n+2}\frac{2n+1+\frac{1}{\sqrt{3}}}{2n+1}\left[f(r_+,\theta_+)+f(r_+,\theta_-)\right]+\\
\frac{2n+1-\frac{1}{\sqrt{3}}}{2n+1}\left[f(r_-,\theta_+)+f(r_-,\theta_-)\right] \qquad (22)
\end{aligned}
$$

3. Rough Surface

The next thing to consider is what happens if the surface over which we are integrating isn't quite a circle. This could happen because the circle has a rough surface or because the area isn't a flat circle, but has some height to it. These are really the same cases, but I'll derive and test the needed metric tensor using the language of the second case.

3.1. Derivation

Let the surface over which I'm integrating be some height z above the circle itself which we'll take to be in the x-y plane. A differential Cartesian area element will have two sides

$$\vec{S}_x = dx\,\hat{x} + \partial z_x\,\hat{z} \tag{23}$$

$$= dx\left(\hat{x} + \frac{\partial z}{\partial x}\hat{z}\right) \tag{24}$$

$$\vec{S}_y = dy\,\hat{y} + \partial z_y\,\hat{z} \tag{25}$$

$$= dy\left(\hat{y} + \frac{\partial z}{\partial y}\hat{z}\right) \tag{26}$$

where ∂z_u represents the variation in z keeping the coordinate u constant. The surface defined by these two vectors is a parallelogram with an area

$$dA = S_x S_y \sin\theta \tag{27}$$

$$= \left|\vec{S}_x \times \vec{S}_y\right| \tag{28}$$

$$= dx\,dy\left|\hat{z} - \frac{\partial z}{\partial y}\hat{y} - \frac{\partial z}{\partial x}\hat{x}\right| \tag{29}$$

$$= dx\,dy\sqrt{1 + \left(\frac{\partial z}{\partial x}\right)^2 + \left(\frac{\partial z}{\partial y}\right)^2} \tag{30}$$

where θ is the angle between $\vec{S}_x$ and $\vec{S}_y$.

3.2. Tests

I checked three cases for reasonableness

3.2.1. Constant z

If z is constant, dA should be

$$dA = dx\,dy\ . \tag{31}$$

Since the partial derivatives are equal to zero, that is indeed the case.

3.2.2. Flat slope

If

$$z = \alpha + \beta x \tag{32}$$

the area is a a translation of a sloped line of length

$$\ell = \sqrt{1+\beta^2} \tag{33}$$

$$dA = dx\,dy\,\sqrt{1+\beta^2}\ . \tag{34}$$

Since

$$\frac{\partial z}{\partial x} = \beta \tag{35}$$

$$\frac{\partial z}{\partial y} = 0 \tag{36}$$

this agrees with Equation 30.

3.3. Half-Sphere

A sphere of radius a has a surface area of

$$A = \frac{4}{3}\pi a^2 \tag{37}$$

so a half sphere would have an area

$$A = \frac{2}{3}\pi a^2\ . \tag{38}$$

Performing an integral over unit circle with a half-spherical dome above it I have

$$z = \sqrt{1-x^2-y^2} \tag{39}$$

$$\frac{\partial z}{\partial x} = -\frac{x}{z} \tag{40}$$

$$\frac{\partial z}{\partial y} = -\frac{y}{z} \tag{41}$$

$$\begin{aligned} dA &= dx\,dy\sqrt{1+\frac{x^2}{z^2}+\frac{y^2}{z^2}} \\ &= z\,dx\,dy\sqrt{x^2+y^2+z^2} \qquad (42) \\ &= z\,dx\,dy\sqrt{x^2+y^2+1-x^2-y^2} \qquad (43) \\ &= z\,dx\,dy\ . \qquad (44) \end{aligned}$$

Switch to polar coordinates to do the integral.

$$dx\,dy = rdr\,d\theta \tag{45}$$
$$z = \sqrt{1 - r^2\cos^2\theta - r^2\sin^2\theta} \tag{46}$$
$$= \sqrt{1 - r^2} \tag{47}$$
$$A = \int_0^1 r\sqrt{1 - r^2}\,dr \int_0^{2\pi} d\theta \tag{48}$$
$$= 2\pi \int_0^1 r\sqrt{1 - r^2}dr \tag{49}$$
$$= \frac{2}{3}\pi \tag{50}$$

This is the expected answer.

4. Accuracy

4.1. Square Rules

The square rules are exact for the following monomial cases:

- any odd powers of x and/or y (rules give 0 since they have the appropriate symmetry
- 1
- x^2
- y^2
- x^2y^2

Translate these into an arc on the n^{th} annulus and the m^{th} section of a circle the inner radius a with $-\frac{a}{2} \le x \le \frac{a}{2}$ and $-\frac{a}{2} \le y \le \frac{a}{2}$.

$$r = \left(n + \frac{1}{2}\right)a + y \tag{51}$$
$$\theta = \frac{2\pi(m + \frac{1}{2} + \frac{x}{a})}{3(2n + 1)} \tag{52}$$
$$= \frac{\pi(2ma + a + 2x)}{3a(2n + 1)} \tag{53}$$

This assumes n and m are indexed starting with 0. Since $r \propto y$ and $\theta \propto x$ these rules should be good for the same powers of r and θ as they are for x and y. Note that these values of x and y are the ones for the underlying square, not the actual x and y coordinates on the arc.

4.2. Circle Rules

For any n and $f = 1$, Equation 19 gives the exact answer of $\pi a^2/3$. For $n = 0$, $m = 0$, and $f = \cos\theta$ the exact answer is

$$\int_0^{2\pi/3} \cos\theta \, d\theta \int_0^a r \, dr = \frac{a^2}{2} \sin\theta|_{\theta=0}^{2\pi/3} \tag{54}$$

$$= \frac{a^2\sqrt{3}}{4} \tag{55}$$

$$= 0.433a^2 \; . \tag{56}$$

The numerical approximation yields

$$\int_0^{2\pi/3} \cos\theta \, d\theta \int_0^a r \, dr \approx a^2 \frac{\pi}{6} \sum \frac{\sqrt{3} \pm 1}{2\sqrt{3}} \cos\left(2\pi \frac{\frac{1}{2} \pm \frac{1}{2\sqrt{3}}}{3}\right) \tag{57}$$

$$\approx \frac{\pi a^2}{6} \sum \cos\left(\frac{\pi(\sqrt{3} \pm 1)}{3\sqrt{3}}\right) \tag{58}$$

$$\approx 0.431a^2 \tag{59}$$

which is a good approximation. For $n = 1$, $m = 0$, $f = \cos\theta$ the exact answer is

$$\int_0^{2\pi/9} \cos\theta \int_a^{2a} r \, dr = \frac{3a^2}{2} \sin\theta|_{\theta=0}^{2\pi/9} \tag{60}$$

$$= 0.9642a^2 \; . \tag{61}$$

The approximation yields

$$\int_0^{2\pi/9} \cos\theta \int_a^{2a} r \, dr \approx \frac{\pi a^2}{6} \sum \cos\left(2\pi \frac{\frac{1}{2} \pm \frac{1}{2\sqrt{3}}}{9}\right) \tag{62}$$

$$\approx \frac{\pi a^2}{6} \sum \cos\left(\frac{\pi(\sqrt{3} \pm 1)}{9\sqrt{3}}\right) \tag{63}$$

$$\approx 0.9641a^2 \tag{64}$$

which agrees to four significant digits. I did some studies with a Julia implementation and made the following observations:

- The square rules on which these rules are based are a product of Gaussian quadrature rules. On each square, calculations are exact for polynomials up to power 3 in x and y. In other words, any of the following functions can be integrated exactly over a single square:

$$f(x, y) = x \tag{65}$$
$$f(x, y) = y \tag{66}$$
$$f(x, y) = C \tag{67}$$
$$f(x, y) = xy \tag{68}$$
$$f(x, y) = x^2 y \tag{69}$$
$$f(x, y) = x^3 y^2 \tag{70}$$
$$f(x, y) = x^3 y^3 \tag{71}$$
$$f(x, y) = (a + bx + cx^3)(d + ey^2 + gy^3) \tag{72}$$

- The integrations will likewise be exact for any power of r from -1 to 2.
- The integrations will be exact for any power of θ from 0 to 3.
- The integrations will converge very quickly for integrations of trigonometric functions which are periodic on a circle.

5. Singular Integrals

For integrals on the same patch, the Greene's function is singular. There are a series of coordinate transformations that make the integrals non-singular so that they can be integrated with Gaussian quadrature product rules. I will outline the series of transformations in this section and give some examples using Mathematica, FORTRAN, Python, and Julia.

5.1. Transforming from Annulus or Pie to Square

The first transformation needed is similar to the one used for the quadrature rule developed in previous sections. Solving Equation 13 for x, we have

$$x = a\left[\frac{\theta}{2\pi}(6n + 3) - m - \frac{1}{2}\right] \tag{73}$$
$$\frac{\partial\theta}{\partial x} = a\frac{6n + 3}{2\pi}\ . \tag{74}$$

Solving Equation 10 for y, we have

$$y = r - a\left(n + \frac{1}{2}\right) \tag{75}$$
$$\frac{\partial r}{\partial y} = 1 \tag{76}$$

giving us a Jacobian matrix J.

$$J = \left\| \begin{array}{cc} \frac{\partial r}{\partial x} & \frac{\partial r}{\partial y} \\ \frac{\partial \theta}{\partial x} & \frac{\partial \theta}{\partial \theta y} \end{array} \right\| \tag{77}$$

$$= \left\| \begin{array}{cc} 0 & 1 \\ \frac{2\pi}{a(6n+3)} & 0 \end{array} \right\| \tag{78}$$

$$= \frac{2\pi}{a(6n+3)} \tag{79}$$

The transformed integration is

$$\theta_{min} = \frac{2\pi m}{3(2n+1)} \tag{80}$$

$$\theta_{max} = \frac{2\pi(m+1)}{3(2n+1)} \tag{81}$$

$$\int_{na}^{(n+1)a} r\,dr \int_{\theta_{min}}^{\theta_{max}} f(r,\theta)\,d\theta = J \int_{-\frac{a}{2}}^{\frac{a}{2}} \left[a\left(n+\frac{1}{2}\right) + y \right] dy \int_{-\frac{a}{2}}^{\frac{a}{2}} f(r(y),\theta(x))\,dx \ , \tag{82}$$

where $r(y)$ is given by Eq. 10 and $\theta(x)$ is given by Eq. 13. To make sure the Jacobian is correct, let's check this formula for the specific case of $n=1$, $m=0$, $f(r,\theta)=1$.

$$\int_{a}^{2a} r\,dr \int_{0}^{2\pi/9} d\theta = \frac{1}{2}\left[(2a)^2 - a^2\right]\frac{2\pi}{9} \tag{83}$$

$$= \frac{\pi a^2}{3} \tag{84}$$

$$\frac{2\pi}{9a}\int_{-\frac{a}{2}}^{\frac{a}{2}} \left[\frac{3}{2}a + y\right] dy \int_{-\frac{a}{2}}^{\frac{a}{2}} dx = \left(\frac{2\pi}{9a}\right)\left(\frac{3}{2}a^2\right)(a) \tag{85}$$

$$= \frac{\pi a^2}{3} \tag{86}$$

Note that the above formulas work for both an annulus ($n > 1$) and a pie-shape ($n = 0$).

5.2. Dividing Square Into Triangles

The next step is to divide the square into triangles with the singular points at a vertex of the triangle. For each singular point, there will be four triangles with the following vertices (listing the singular vertex first).

1. $\left(\pm a/2\sqrt{3}, \pm a/2\sqrt{3}\right)$, $(-a/2, -a/2)$, $(-a/2, a/2)$
2. $\left(\pm a/2\sqrt{3}, \pm a/2\sqrt{3}\right)$, $(-a/2, a/2)$, $(a/2, a/2)$
3. $\left(\pm a/2\sqrt{3}, \pm a/2\sqrt{3}\right)$, $(a/2, a/2)$, $(a/2, -a/2)$
4. $\left(\pm a/2\sqrt{3}, \pm a/2\sqrt{3}\right)$, $(a/2, -a/2)$, $(-a/2, -a/2)$

Fig. 2 shows the triangles formed with the upper left singular point as a vertex.

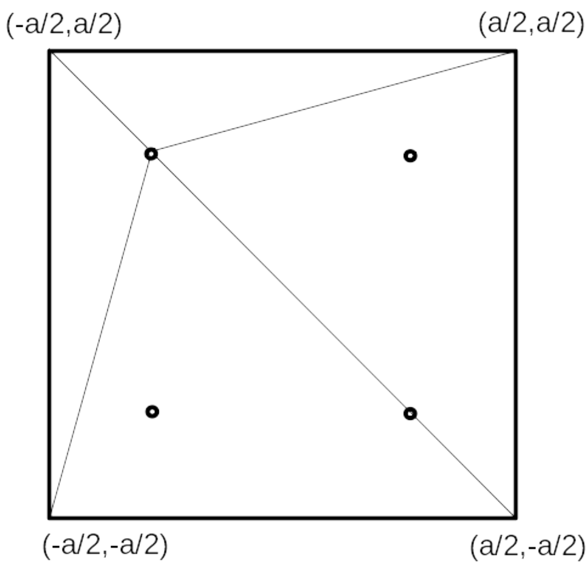

Figure 2: Triangles formed with the upper left singular point as a vertex. $(-a/2\sqrt{3}, a/2\sqrt{3})$.

5.3. Transforming to Right Triangles

The above triangles are not right triangles. The Duffy transformation is in terms of a right triangle with

$$0 \leq x \leq 1 \tag{87}$$

$$0 \leq y \leq 1 \ . \tag{88}$$

To do this, we first translate the axes to the origin.

$$x' = x \mp \frac{a}{2\sqrt{3}} \tag{89}$$

$$y' = y \mp \frac{a}{2\sqrt{3}} \tag{90}$$

This gives us the following four triangles.

1. $(0,0)$, $(-a/2 \mp a/2\sqrt{3}, -a/2 \mp a/2\sqrt{3})$, $(-a/2 \mp a/2\sqrt{3}, a/2 \mp a/2\sqrt{3})$
2. $(0,0)$, $(-a/2 \mp a/2\sqrt{3}, a/2 \mp a/2\sqrt{3})$, $(a/2 \mp a/2\sqrt{3}, a/2 \mp a/2\sqrt{3})$
3. $(0,0)$, $(a/2 \mp a/2\sqrt{3}, a/2 \mp a/2\sqrt{3})$, $(a/2 \mp a/2\sqrt{3}, -a/2 \mp a/2\sqrt{3})$
4. $(0,0)$, $(a/2 \mp a/2\sqrt{3}, -a/2 \mp a/2\sqrt{3})$, $(-a/2 \mp a/2\sqrt{3}, -a/2 \mp a/2\sqrt{3})$

Let the vertices of each triangle be in the order listed above. This means the origin, where the singularity is, will be the first vertex of each triangle. Next, we do a linear transformation to a new coordinate system where each leg of the right triangle is parallel to a coordinate axis. This means the right angle is at the transformed second vertex where $(x, y) = (1, 0)$. Let the coordinates of the second vertex of the triangle be

$$x' = p_2 \tag{91}$$
$$y' = q_2 \tag{92}$$

and the coordinates of the third vertex be

$$x' = p_3 \tag{93}$$
$$y' = q_3 \ . \tag{94}$$

Then the transformation is as follows.

$$\begin{pmatrix} x'' \\ y'' \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} x' \\ y' \end{pmatrix} \tag{95}$$
$$\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} p_2 \\ q_2 \end{pmatrix} \tag{96}$$
$$\begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} p_3 \\ q_3 \end{pmatrix} \tag{97}$$

Eq. 96 and 97 can be combined into a single martrix equation. This gives the transformation matrix to transform the primed coordinates to the double primed coordinates.

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} p_2 & p_3 \\ q_2 & q_3 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \tag{98}$$

At this point, I could explicitly solve for the matrix A, but it turns out what I need in practice is the inverse of A which I'll call B so that I can go from the double primed coordinates back to the primed coordinates. Since

$$BA = 1, \tag{99}$$

I can left multiply both sides of Eq. 98 by B to get

$$BA \begin{pmatrix} p_2 & p_3 \\ q_2 & q_3 \end{pmatrix} = B \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \tag{100}$$
$$B = \begin{pmatrix} p_2 & p_3 \\ q_2 & q_3 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}^{-1} \tag{101}$$
$$= \begin{pmatrix} p_2 & p_3 \\ q_2 & q_3 \end{pmatrix} \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix} \tag{102}$$
$$= \begin{pmatrix} p_2 & p_3 - p_2 \\ q_2 & q_3 - q_2 \end{pmatrix} \tag{103}$$

5.4. Alternate Transformation to Right Triangles

The matrices B can be calculated in a more straightforward way. This will also serve as a check of the above calculations. As we do these transformations, we number the points using the same order as the closest second vertex of the four triangles from Section 5.2.

The inverse transformations can be found directly from Equations 95 through 97.

$$\begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \begin{pmatrix} x'' \\ y'' \end{pmatrix} = \begin{pmatrix} x' \\ y' \end{pmatrix} \tag{104}$$
$$\begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} p_2 \\ q_2 \end{pmatrix} \tag{105}$$
$$\begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} p_3 \\ q_3 \end{pmatrix} \tag{106}$$

These equations can be solved simply for B in terms of the known variables p and q for the vertices. Multiplying the matrices in Equations 105 and 106 explicitly gives us the following four equations.

$$B_{11} = p_2 \tag{107}$$
$$B_{21} = q_2 \tag{108}$$
$$B_{11} + B_{12} = p_3 \tag{109}$$
$$B_{21} + B_{22} = q_3 \tag{110}$$

These are straightforward to solve for B.

$$B = \begin{pmatrix} p_2 & p_3 - p_2 \\ q_2 & q_3 - q_2 \end{pmatrix} \tag{111}$$

This agrees with Eq. 103.

5.5. Transformation Matrices

For the four different triangles,

$$2\sqrt{3}p_2 = a(-\sqrt{3} \mp 1, -\sqrt{3} \mp 1, \sqrt{3} \mp 1, \sqrt{3} \mp 1) \tag{112}$$
$$2\sqrt{3}p_3 = a(-\sqrt{3} \mp 1, \sqrt{3} \mp 1, \sqrt{3} \mp 1, -\sqrt{3} \mp 1) \tag{113}$$
$$2\sqrt{3}q_2 = a(-\sqrt{3} \mp 1, \sqrt{3} \mp 1, \sqrt{3} \mp 1, -\sqrt{3} \mp 1) \tag{114}$$
$$2\sqrt{3}q_3 = a(\sqrt{3} \mp 1, \sqrt{3} \mp 1, -\sqrt{3} \mp 1, -\sqrt{3} \mp 1) \tag{115}$$
$$p_3 - p_2 = (0, 1, 0, -1) \tag{116}$$
$$q_3 - q_2 = (1, 0, -1, 0) \tag{117}$$

The $\mp$ signs in the above equations depend on the singular point s which can run from 1 to 4. The sign for the various expressions for p and q are summarized Table 1.

singular point	p	q
1	+	+
2	+	-
3	-	-
4	-	+

Table 1: Signs for $\mp$ with the various singular points in Eq. 112 through Eq. 115.

5.5.1. B Matrices

These give the following values for the B matrices. The subscript on the matrix specifies which triangle it is for.

$$B_1 = \frac{a}{2\sqrt{3}} \begin{pmatrix} -\sqrt{3} \mp 1 & 0 \\ -\sqrt{3} \mp 1 & 2\sqrt{3} \end{pmatrix} \tag{118}$$

$$B_2 = \frac{a}{2\sqrt{3}} \begin{pmatrix} -\sqrt{3} \mp 1 & 2\sqrt{3} \\ \sqrt{3} \mp 1 & 0 \end{pmatrix} \tag{119}$$

$$B_3 = \frac{a}{2\sqrt{3}} \begin{pmatrix} \sqrt{3} \mp 1 & 0 \\ \sqrt{3} \mp 1 & -2\sqrt{3} \end{pmatrix} \tag{120}$$

$$B_4 = \frac{a}{2\sqrt{3}} \begin{pmatrix} \sqrt{3} \mp 1 & -2\sqrt{3} \\ -\sqrt{3} \mp 1 & 0 \end{pmatrix} \tag{121}$$

Let the superscript in B designate the singular point (see Fig. 2 and Table 1), where the first singular point is in the lower-left corner and they are numbered going clockwise. The subscript refers to the corners used in the triangle as above. The first triangle uses the lower left corner at $(-a/2, -a/2)$ and the upper left corner at $(-a/2, a/2)$. Subsequent triangles are with the corners rotated in a clockwise direction. Note that this convention involves an improper rotation, switching the order of the corners of the triangle as it is traversed in a clockwise direction. With this convention, the more explicit values for B as as follows.

$$B_1^{(1)} = \frac{a}{2\sqrt{3}} \begin{pmatrix} -\sqrt{3} + 1 & 0 \\ -\sqrt{3} + 1 & 2\sqrt{3} \end{pmatrix} \tag{122}$$

$$B_1^{(2)} = \frac{a}{2\sqrt{3}} \begin{pmatrix} -\sqrt{3} + 1 & 0 \\ -\sqrt{3} - 1 & 2\sqrt{3} \end{pmatrix} \tag{123}$$

$$B_1^{(3)} = \frac{a}{2\sqrt{3}} \begin{pmatrix} -\sqrt{3} - 1 & 0 \\ -\sqrt{3} - 1 & 2\sqrt{3} \end{pmatrix} \tag{124}$$

$$B_1^{(4)} = \frac{a}{2\sqrt{3}} \begin{pmatrix} -\sqrt{3} - 1 & 0 \\ -\sqrt{3} + 1 & 2\sqrt{3} \end{pmatrix} \tag{125}$$

$$B_2^{(1)} = \frac{a}{2\sqrt{3}} \begin{pmatrix} -\sqrt{3}+1 & 2\sqrt{3} \\ \sqrt{3}+1 & 0 \end{pmatrix} \tag{126}$$

$$B_2^{(2)} = \frac{a}{2\sqrt{3}} \begin{pmatrix} -\sqrt{3}+1 & 2\sqrt{3} \\ \sqrt{3}-1 & 0 \end{pmatrix} \tag{127}$$

$$B_2^{(3)} = \frac{a}{2\sqrt{3}} \begin{pmatrix} -\sqrt{3}-1 & 2\sqrt{3} \\ \sqrt{3}-1 & 0 \end{pmatrix} \tag{128}$$

$$B_2^{(4)} = \frac{a}{2\sqrt{3}} \begin{pmatrix} -\sqrt{3}-1 & 2\sqrt{3} \\ \sqrt{3}+1 & 0 \end{pmatrix} \tag{129}$$

$$B_3^{(1)} = \frac{a}{2\sqrt{3}} \begin{pmatrix} \sqrt{3}+1 & 0 \\ \sqrt{3}+1 & -2\sqrt{3} \end{pmatrix} \tag{130}$$

$$B_3^{(2)} = \frac{a}{2\sqrt{3}} \begin{pmatrix} \sqrt{3}+1 & 0 \\ \sqrt{3}-1 & -2\sqrt{3} \end{pmatrix} \tag{131}$$

$$B_3^{(3)} = \frac{a}{2\sqrt{3}} \begin{pmatrix} \sqrt{3}-1 & 0 \\ \sqrt{3}-1 & -2\sqrt{3} \end{pmatrix} \tag{132}$$

$$B_3^{(4)} = \frac{a}{2\sqrt{3}} \begin{pmatrix} \sqrt{3}-1 & 0 \\ \sqrt{3}+1 & -2\sqrt{3} \end{pmatrix} \tag{133}$$

$$B_4^{(1)} = \frac{a}{2\sqrt{3}} \begin{pmatrix} \sqrt{3}+1 & -2\sqrt{3} \\ -\sqrt{3}+1 & 0 \end{pmatrix} \tag{134}$$

$$B_4^{(2)} = \frac{a}{2\sqrt{3}} \begin{pmatrix} \sqrt{3}+1 & -2\sqrt{3} \\ -\sqrt{3}-1 & 0 \end{pmatrix} \tag{135}$$

$$B_4^{(3)} = \frac{a}{2\sqrt{3}} \begin{pmatrix} \sqrt{3}-1 & -2\sqrt{3} \\ -\sqrt{3}-1 & 0 \end{pmatrix} \tag{136}$$

$$B_4^{(4)} = \frac{a}{2\sqrt{3}} \begin{pmatrix} \sqrt{3}-1 & -2\sqrt{3} \\ -\sqrt{3}+1 & 0 \end{pmatrix} \tag{137}$$

5.5.2. Jacobians

Since the transformation is a linear one, the Jacobian for the transformation is the absolute value of the determinant of the B matrix. Here are the absolute values of the associated Jacobian determinants

$$J_1 = a^2 \frac{3 \pm \sqrt{3}}{6} \tag{138}$$

$$J_2 = a^2 \frac{3 \mp \sqrt{3}}{6} \tag{139}$$

$$J_3 = a^2 \frac{3 \mp \sqrt{3}}{6} \tag{140}$$

$$J_4 = a^2 \frac{3 \pm \sqrt{3}}{6} \tag{141}$$

Using the same superscript and subscript convention as for the B matrices, the Jacobians are:

$$J_1^{(1)} = a^2 \frac{3-\sqrt{3}}{6} \tag{142}$$

$$J_1^{(2)} = a^2 \frac{3-\sqrt{3}}{6} \tag{143}$$

$$J_1^{(3)} = a^2 \frac{3+\sqrt{3}}{6} \tag{144}$$

$$J_1^{(4)} = a^2 \frac{3+\sqrt{3}}{6} \tag{145}$$

$$J_2^{(1)} = a^2 \frac{3+\sqrt{3}}{6} \tag{146}$$

$$J_2^{(2)} = a^2 \frac{3-\sqrt{3}}{6} \tag{147}$$

$$J_2^{(3)} = a^2 \frac{3-\sqrt{3}}{6} \tag{148}$$

$$J_2^{(4)} = a^2 \frac{3+\sqrt{3}}{6} \tag{149}$$

$$J_3^{(1)} = a^2 \frac{3+\sqrt{3}}{6} \tag{150}$$

$$J_3^{(2)} = a^2 \frac{3+\sqrt{3}}{6} \tag{151}$$

$$J_3^{(3)} = a^2 \frac{3-\sqrt{3}}{6} \tag{152}$$

$$J_3^{(4)} = a^2 \frac{3-\sqrt{3}}{6} \tag{153}$$

$$J_4^{(1)} = a^2 \frac{3-\sqrt{3}}{6} \tag{154}$$

$$J_4^{(2)} = a^2 \frac{3+\sqrt{3}}{6} \tag{155}$$

$$J_4^{(3)} = a^2 \frac{3+\sqrt{3}}{6} \tag{156}$$

$$J_4^{(4)} = a^2 \frac{3-\sqrt{3}}{6}. \tag{157}$$

5.5.3. Checking Jacobian Determinant

I'll check the Jacobian determinant by integrating a unit function as before assuming a (non-existent) singularity at $(x, y) = \left(a/2\sqrt{3}, a/2\sqrt{3}\right)$. I will use a superscript to designated the singular point when it matters from now on. The superscript in this case is 3. The square has an area of a^2 before being subdivided. The translation of the center doesn't change the area. The four triangles after transformation have areas of

$$A_t = \frac{J_i}{2} \, . \tag{158}$$

The four triangles in this case have respective Jacobian determinants of

$$J_1^{(3)} = \frac{(3+\sqrt{3})a^2}{6} \tag{159}$$

$$J_2^{(3)} = \frac{(3-\sqrt{3})a^2}{6} \tag{160}$$

$$J_3^{(3)} = \frac{(3-\sqrt{3})a^2}{6} \tag{161}$$

$$J_4^{(3)} = \frac{(3+\sqrt{3})a^2}{6} \tag{162}$$

whose sum is

$$J_1^{(3)} + J_2^{(3)} + J_3^{(3)} + J_4^{(3)} = 2a^2 \, . \tag{163}$$

This gives a total area of a^2 which agrees with the untransformed square.

5.5.4. B Unit Tests

One way to test B and J in unit tests is to integrate over the square before transformation and compare it to the integrals after the transformation.

Before Transformation Before the transformation, the integration is over a square with sides of length a, but the origin translated to a singular point. If the integrand

$$f(x', y') = 1 \tag{164}$$

then the integral over the square is just the area of the square. This gives us a test of the Jacobian as noted in Sec. 5.5.3. Letting

$$f(x', y') = x' \tag{165}$$

$$f(x', y') = y' \tag{166}$$

$$f(x', y') = x'y' \tag{167}$$

gives us a check of the B matrix. Before the transformation using the singular point (b, c)

$$\int_{square} x' \, dx' \, dy' = a \int (x' - b) \, dx' \tag{168}$$

$$= -ba^2. \tag{169}$$

After the transformation the area of one triangle is

$$A = J\int_0^1 du \int_0^u (B_{11}u + B_{12}v)\, dv \tag{170}$$
$$= J\int_0^1 \left(B_{11}u + \frac{1}{2}B_{12}u^2\right) du \tag{171}$$
$$= J\int_0^1 \left(B_{11} + \frac{1}{2}B_{12}\right) u^2\, du \tag{172}$$
$$= \left(\frac{1}{3}B_{11} + \frac{1}{6}B_{12}\right) J. \tag{173}$$

Summing this over the four squares should give the same result as Eq. 169. Similarly, we can use

$$f(x', y') = y' \tag{174}$$

as the integrand.

$$\int_{square} y'\, dx'\, dy' = a\int (y' - c)\, dy' \tag{175}$$
$$= -ca^2. \tag{176}$$

The same integral after the transformation is

$$A = J\int_0^1 du \int_0^u (B_{21}u + B_{22}v)\, dv \tag{177}$$
$$= J\int_0^1 \left(B_{21}u^2 + \frac{1}{2}B_{22}u^2\right) du \tag{178}$$
$$= J\int_0^1 \left(B_{21} + \frac{1}{2}B_{22}\right) u^2\, du \tag{179}$$
$$= \left(\frac{1}{3}B_{21} + \frac{1}{6}B_{22}\right) J. \tag{180}$$

Summing this over the four squares should give the same result as Eq. 176. A final check on B is to let

$$f(x', y') = x'y'. \tag{181}$$

Then the integral before the transformation is

$$\int_{square} x'y'\, dx'\, dy' = \int (x' - b)\, dx' \int (y' - c)\, dy' \tag{182}$$
$$= bca^2 \tag{183}$$

After the transformation the the area is

$$A = J\int_0^1 du \int_0^u (B_{11}u + B_{12}v)(B_{21}u + B_{22}v)dv \tag{184}$$

$$= J\int_0^1 u^3 \left[B_{11}B_{21} + \frac{1}{2}(B_{12}B_{21} + B_{11}B_{22}) + \frac{1}{3}B_{12}B_{22}\right) \tag{185}$$

$$= \left(\frac{B_{11}B_{21}}{4} + \frac{B_{12}B_{21} + B_{11}B_{22}}{8} + \frac{B_{12}B_{22}}{12}\right) J. \tag{186}$$

This should equal Eq. 183.

5.6. Duffy Transformation

The next step is to turn each of the right triangles into squares. The x coordinate will transform directly into a new u coordinate. The new v coordinate will be y''/x''. The old domains for the triangles were

$$0 \le x'' \le 1 \tag{187}$$

$$0 \le y'' \le x'' \,. \tag{188}$$

With the transformation

$$x'' = u \tag{189}$$

$$y'' = uv \tag{190}$$

$$u = x'' \tag{191}$$

$$v = \frac{y''}{x''} \tag{192}$$

the Jacobian determinant is

$$J = \left\| \begin{array}{cc} \frac{\partial x''}{\partial u} & \frac{\partial x''}{\partial v} \\ \frac{\partial y''}{\partial u} & \frac{\partial y''}{\partial v} \end{array} \right\| \tag{193}$$

$$= \left| \begin{array}{cc} 1 & 0 \\ v & u \end{array} \right| \tag{194}$$

$$= u \,. \tag{195}$$

Check this transformation with integrating a function $f(x'', y'') = 1$.

$$\int_0^1 dx'' \int_0^{x''} dy'' = \int_0^1 x''\, dx'' \tag{196}$$

$$= \frac{1}{2} \tag{197}$$

$$\int_0^1 u\, du \int_0^1 dv = \int_0^1 u\, du \tag{198}$$

$$= \frac{1}{2} \tag{199}$$

5.7. Summary

To integrate $rf(r,\theta)$ over the arc with n specifying the arc number (starting with 0) and m specifying the segment within the arc (starting at 0) use the following substitutions.

$$r = y + a\left(n + \frac{1}{2}\right) \tag{200}$$

$$\theta = 2\pi \frac{a\left(m+\frac{1}{2}\right)+x}{3a(2n+1)} \tag{201}$$

$$x = x' \pm \frac{a}{2\sqrt{3}} \tag{202}$$

$$y = y' \pm \frac{a}{2\sqrt{3}} \tag{203}$$

$$x' = B_{11}x'' + B_{12}y'' \tag{204}$$

$$y' = B_{21}x'' + B_{22}y'' \tag{205}$$

$$x'' = u \tag{206}$$

$$y'' = uv \tag{207}$$

Following down the chain, this lets you compute $r(u,v)$ and $\theta(u,v)$. Then you just substitute, multiply by the Jacobians, and integrate.

$$\int rf(r,\theta)\,dr\,d\theta = \left[\frac{2\pi}{a(6n+3)}\right] J_i^{(j)} \int_0^1 u\,du \int_0^1 r(u,v)f(r(u,v),\theta(u,v))\,dv \tag{208}$$

5.8. Concrete Example

This is an example of how to implement these formulas in a specific case. I will consider the case where the singularity is singular point 2 and I want to integrate triangle number 3.

$$(x,y) = \left(-\frac{a}{2\sqrt{3}}, \frac{a}{2\sqrt{3}}\right). \tag{209}$$

To avoid too many complications, I'll let $f(r,\theta) = G(\rho)$, ignoring any roughness on the surface. I'll define

$$G(\rho) = \frac{e^{ik\rho}}{4\pi\rho} \tag{210}$$

$$\rho = \sqrt{(r\cos\theta - r'\cos\theta')^2 + (r\sin\theta - r'\sin\theta')^2} \tag{211}$$

$$= \sqrt{r^2 + r'^2 - 2rr'\cos(\theta-\theta')} \tag{212}$$

where

$$(r',\theta') = \left(a\left[\frac{1}{2\sqrt{3}} + n + \frac{1}{2}\right], 2\pi\frac{m+\frac{1}{2}-\frac{1}{2\sqrt{3}}}{6n+3}\right) \tag{213}$$

$$= \left(a\frac{2\sqrt{3}n+\sqrt{3}+1}{2\sqrt{3}}, 2\pi\frac{2\sqrt{3}m+\sqrt{3}-1}{6\sqrt{3}(2n+1)}\right) \tag{214}$$

are the coordinates of the singularity (using Eq. 200 and Eq. 201). Substituting Eq. 202 into Eq. 201 and Eq. 203 into Eq. 200,

$$r = r' + y' \tag{215}$$

$$= y' + \frac{a}{2\sqrt{3}} + a\left(n + \frac{1}{2}\right) \tag{216}$$

$$\theta = \theta' + \frac{2\pi x'}{3a(2n+1)} \tag{217}$$

$$= 2\pi \frac{a\left(m + \frac{1}{2}\right) + x' - \frac{a}{2\sqrt{3}}}{3a(2n+1)}\ . \tag{218}$$

Substituting Eq. 215 and Eq. 217 into Eq. 212,

$$\rho = \sqrt{(r' + y')^2 + r'^2 - 2(r' + y')r' \cos\left(\frac{2\pi x'}{3a(2n+1)}\right)}. \tag{219}$$

By substituting Eq. 206 into Eq. 204 and Eq. 207 into Eq. 205,

$$x' = B_{11}u + B_{12}uv \tag{220}$$

$$y' = B_{21}u + B_{22}uv, \tag{221}$$

showing that $\rho = 0$ when $u = 0$. Next, substitute Eq. 220 and Eq. 221 into Eq. 215 and Eq. 217.

$$r = r' + B_{21}u + B_{22}uv \tag{222}$$

$$\theta = \theta' + \frac{2\pi(B_{11}u + B_{12}uv)}{3a(2n+1)} \tag{223}$$

For the third triangle, the inverse transformation matrix is from Eq. 131 and Jacobian from Eq. 151. Substituting Eq. 131 into Eq. 222 and 223,

$$r = r' + B_{21}u + B_{22}uv \tag{224}$$

$$= r' + \frac{a}{2\sqrt{3}}[(\sqrt{3} - 1)u - 2\sqrt{3}uv] \tag{225}$$

$$= r' + \frac{a[(3 - \sqrt{3})u - 6uv]}{6} \tag{226}$$

$$\theta = \theta' + \frac{2\pi(B_{11}u + B_{12}uv)}{3a(2n+1)} \tag{227}$$

$$= \theta' + \frac{2\pi u}{3a(2n+1)} \frac{a}{2\sqrt{3}}(\sqrt{3} + 1) \tag{228}$$

$$= \theta' + \frac{\pi(3 + \sqrt{3})}{9(2n+1)}u. \tag{229}$$

We can now compute the desired integral for this triangle using the Jacobians from Eq. 79 and Eq.151.

$$\int rG(\rho)\, dr\, d\theta = \left[\frac{2\pi}{a(6n+3)}\right]\left(a^2 \frac{3 + \sqrt{3}}{6}\right) \int_0^1 u\, du \int_0^1 r(u,v) G(\rho(u,v))\, dv \tag{230}$$

5.9. Computer Implementations

I have implemented these algorithms in FORTRAN, Mathematica, Python, and Julia. They will be hepful for computing same-patch matrix elements involving a singular kernel.

5.9.1. FORTRAN

I have implemented the above algorithms in two modules, `patch.f95` and `alt_patch.f95`. I tested integrating over the patches using the pFUnit unit testing framework in the file `test_patch.pf`. These three files are included in their entirety in the appendix, but I'll discuss some details of each here.

The module `patch` in the file `patch.f95` has the `patch_par` structure which is implemented as a class. It's primary purpose is to compute and store the B matrix and J vector.

The module `alt_patch` in the file `alt_patch.f95` is similar, but computes B and J directly using the alternative formulas from Sec. 5.4. The module has an identical interface to the patch module so that the two can be interchanged and compared for testing purposes.

The file `test_patch.pf` has the unit tests for the FORTRAN code. The subroutine `Parameters` tests the `struct` initialization for `patch` or `alt_patch` depending on which `use` statement is uncommented. The test in subroutine `altB` explicitly tests the elements of the `alt_patch` B matrix construction. The test in subroutine `altJ` explicitly tests the elements of the `alt_patch` J vector construction.

The tests `patchPar` and `PatchBJ` compare the parameters and construction of B and J from the `patch` and `alt_patch` modules.

The function `constInt` compares the Duffy integration of a unit Greene function over a patch to the answer computed using Mathematica. Similarly, `GreeneInt` compares Mathematica and FORTRAN computations of the Greene's function over the patch area.

5.9.2. Mathematica

The Mathematica implementation was used to check both the math and numerical values of the other implementations. Here is the code for computing the B matrices. The first argument is the triangle number. The second argument is the singular point number. The last argument is the radius of the distance between annular rings.

```
B[tri_,sp_,a_]:=Module[{p, q,fct=a/(2*Sqrt[3])},
  {p,q}={{1,1},{1,-1},{-1,-1},{-1,1}}[[sp]];
  fct*Switch[tri,
    1,{{Sqrt[3]-p,0},{Sqrt[3]-q,2Sqrt[3]}},
    2,{{Sqrt[3]-p, 2Sqrt[3]},{Sqrt[3]+q,0}},
    3, {{Sqrt[3]+p,0},{Sqrt[3]+q,-2Sqrt[3]}},
    4,{{Sqrt[3]+p,-2Sqrt[3]},{Sqrt[3]-q,0}}]
  ]
```

I checked for an exact match when `tri=3` and `sp=2`. Here is a simple, but inefficient calculation of J.

```
J[tri_,sp_,a_]:=Abs[Det[B[tri,sp,a]]]
```

I tested this for the same case as B. Here are two examples of integrating 1.0 over a single patch, comparing NIntegrate and the exact answer.

```
a=1.0;
n=4;
m=5;
nint=NIntegrate[\[Rho],{\[Rho],a n,a(n+1)},{\[Theta],
  2Pi m/(3(2 n+1)),2 Pi (m      +1)/((3(2n+1)))}]
exact = Pi a^2/3
```

The two results both 1.0462. Here is another example with different parameters.

```
a=1.5;
n=2;
m=1;
nint=NIntegrate[\[Rho],{\[Rho],a n,a(n+1)},{\[Theta],
  2Pi m/(3(2 n+1)),2 Pi (m+1)/((3(2n+1)))}]
exact = Pi a^2/3
```

The two calculations both give 2.35619. Here is the code for computing the Greene's function.

```
G[r_,\[Theta]_,rp_,\[Theta]p_, k_]:= Module[{\[Rho]},
  \[Rho] = Sqrt[r^2+rp^2-2r rp Cos[\[Theta]-\[Theta]p]];
  Exp[I k \[Rho]]/(4 Pi \[Rho])
]
```

The arguments are r, θ, r', θ', and the wave vector k. Here is the code to evaluate the integral of the Greene's function over an entire patch with singular point number 3.

```
a=1.5;
n=2;
m=1;
k = 2 Pi;
rsp = a/(2Sqrt[3])+a(n+1/2);
tsp = 2 Pi (a(m+1/2)+a/(2Sqrt[3])/(3a(2n+1)));
NIntegrate[rp G[rsp, tsp, rp, \[Theta]p, k],
  {rp,a n,a(n+1)},{\[Theta]p,2Pi m/(3(2 n+1)),
  2 Pi (m+1)/((3(2n+1)))},
  Exclusions->{{rsp,tsp}}]
```

It caculated the exact integral to be $-0.00487878 - 0.0014809i$.

5.9.3. Python

The python code in Appendix B includes the code `cint.py` for integrating a function `f` over a circle of radius `r` with `n` rings. It has unit tests a constant function and a function which is linear in r and θ.

The file `patch.py` is for computing the transformation parameters to enable numerical integration. It includes test cases for checking initialization parameters and computing B and J. The computations of B and J are compared to the FORTRAN code which was verified by comparison to Mathematica calculations.

The Duffy integration is implemented and tested in the `duffy.py` file. It is tested with non-singular functions to compare the results with an exact integration.

5.9.4. Julia

The code in Appendix A demonstrates how to implement thee algorithms in Julia. Sec.A.1 has the code from `cint.jl` for numerically integrating a non-singular function defined on a circle.

6. Nyström Application

The purpose of deriving these integration rules it to apply them to the solution of the integral equation

$$V(\vec{x}) = \int S(\vec{x}')J(\vec{x}')G(\vec{x},\vec{x}')\,d\vec{x'} \tag{231}$$

where the integration is over the mirror surface. The function $S(\vec{x}')$ is the surface metric consisting of the square root in Eq. 30. In the solution of Eq. 231, $V(\vec{x}')$, $S(\vec{x}')$, and $G(\vec{x},\vec{x}')$ are known and one wishes to compute $J(\vec{x}')$ on the surface. This can be readily done using the Nyström method if $G(\vec{x},\vec{x'})$ is finite at all of the discrete integration points developed in Sec. 2. The rules in that section could be utilitzed to replace the surface integral with a sum.

$$\int S(\vec{x}')J(\vec{x}')G(\vec{x},\vec{x}')\,d\vec{x'} \approx \sum_j S(\vec{x}_j)J(\vec{x}_j)G(\vec{x},\vec{x}_j)w_j \ , \tag{232}$$

where w_j is the product of the factors multiplying the integrand in the quadrature rule and $\vec{x}_j$) are the quadrature points from the quadrature rule. The Nyström technique involves substituting Eq. 232 into Eq. 231 and then evaluating the sum at the points $\vec{x} \in \{\vec{x}_i\}$.

$$V(\vec{x}_i) = \sum_j S(\vec{x}_j)J(\vec{x}_j)G(\vec{x}_i,\vec{x}_j)w_j \ , \tag{233}$$

This results in p equations and p unknown $J(\vec{x}_i)$ values where p is the number of quadrature points. If we let

$$V_i = V(\vec{x}_i) \tag{234}$$

$$J_j = V(\vec{x}_j) \tag{235}$$

$$Z_{ij} = S(\vec{x}_j)G(\vec{x}_i, \vec{x}_j)w_j \tag{236}$$

then Eq. 233 becomes the matrix equation

$$V_i = \sum_j Z_{ij} J_j \; . \tag{237}$$

The equation will yield accurate answers if the matrix elements Z_{ij} are reasonably well approximated by a linear combination of the basis functions discussed in Sec. 4 for which the quadrature rules are exact.

This approximation fails miserably, of course, when the function $G(\vec{x}, \vec{x}')$ is singular. This happens for the diagonal elements of the impedance matrix Z_{ii}. In this case, we need to apply the quadrature formulas from Sec. 5. In the case of singular integrands, we require that the quadrature rules be exact for integrating a monomial times the singular kernel rather than just the monomial (as in the case points from different patches). In other words, we approximate

$$\int_{patch} K(\vec{x}')f(\vec{x}')\, d\vec{x}' \approx \sum_j f(\vec{x}_j)w_j \tag{238}$$

with the requirement that the w_j be chosen so that

$$\sum_j w_j = \int K(\vec{x}')\, d\vec{x}' \tag{239}$$

$$\sum_j x_j w_j = \int x' K(\vec{x}')\, d\vec{x}' \tag{240}$$

$$\sum_j y_j w_j = \int y' K(\vec{x}')\, d\vec{x}' \tag{241}$$

$$\sum_j x_j y_j w_j = \int x' y' K(\vec{x}')\, d\vec{x}' \; . \tag{242}$$

This sucks the singular kernel into the quadrature rule and alleviates having to evalulate it at the singular point. The numerical integrals in Eq. 239 through Eq. 242 can be carried out using the techniques of Sec. 5. Then the w_j values can be computed from the coupled equations Eq. 239 through Eq. 242 using the known values of the integrals and the patch integration points (x_j, y_j).

6.1. Application on Square Patch

Equations 239 through 242 can be solved more explicitly by writing them in matrix form. Let

$$K_1 = \int K(\vec{x}')\, d\vec{x}' \tag{243}$$

$$K_2 = \int x' K(\vec{x}')\, d\vec{x}' \tag{244}$$

$$K_3 = \int y' K(\vec{x}')\, d\vec{x}' \tag{245}$$

$$K_4 = \int x' y' K(\vec{x}')\, d\vec{x}' \,. \tag{246}$$

Then the matrix equation is

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ x_1 & x_2 & x_3 & x_4 \\ y_1 & y_2 & y_3 & y_4 \\ x_1y_1 & x_2y_2 & x_3y_3 & x_4y_4 \end{pmatrix} \begin{pmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \end{pmatrix} = \begin{pmatrix} K_1 \\ K_2 \\ K_3 \\ K_4 \end{pmatrix} \tag{247}$$

This has a relatively simple solution if we substitute the values of x_j and y_j as the singular points on the square patch outlined in Sec. 2.

$$x_1 = -\frac{a}{2\sqrt{3}} \tag{248}$$

$$x_2 = -\frac{a}{2\sqrt{3}} \tag{249}$$

$$x_3 = \frac{a}{2\sqrt{3}} \tag{250}$$

$$x_4 = \frac{a}{2\sqrt{3}} \tag{251}$$

$$y_1 = -\frac{a}{2\sqrt{3}} \tag{252}$$

$$y_2 = \frac{a}{2\sqrt{3}} \tag{253}$$

$$y_3 = \frac{a}{2\sqrt{3}} \tag{254}$$

$$y_4 = -\frac{a}{2\sqrt{3}} \tag{255}$$

Plugging these into the matrix and solving with Mathematica,

$$\begin{pmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \end{pmatrix} = \begin{pmatrix} \frac{K_1}{4} - \frac{\sqrt{3}K_2}{2a} - \frac{\sqrt{3}K_3}{2a} + \frac{3K_4}{a^2} \\ \frac{K_1}{4} - \frac{\sqrt{3}K_2}{2a} + \frac{\sqrt{3}K_3}{2a} - \frac{3K_4}{a^2} \\ \frac{K_1}{4} + \frac{\sqrt{3}K_2}{2a} + \frac{\sqrt{3}K_3}{2a} + \frac{3K_4}{a^2} \\ \frac{K_1}{4} + \frac{\sqrt{3}K_2}{2a} - \frac{\sqrt{3}K_3}{2a} - \frac{3K_4}{a^2} \end{pmatrix} \tag{256}$$

6.2. Choice of Singular Kernel

I'm not sure of the best choice for breaking up the integrand from Eq. 231 into the singular kernel K and the smooth function f. The most efficient choice would be to have

$$K(\vec{x}') = G(\vec{x}, \vec{x}') \tag{257}$$
$$f(\vec{x}') = S(\vec{x}')J(\vec{x}') \ . \tag{258}$$

Since the Greene's function G only depends on the distrance between the quadrature points, the numerical integrals will only need to be calculated for one annular patch in each ring.

On the other hand, the singular integrals will be done with high accuracy and the resulting quadrature rule is only accurate to first order in x and y. Letting

$$K(\vec{x}') = G(\vec{x}, \vec{x}')S(\vec{x}') \tag{259}$$
$$f(\vec{x}') = J(\vec{x}') \tag{260}$$

leaves the surface roughness inside the precise numerical integration. This may improve accuracy for a given patch size since J is expected to be smoother than SJ. I will check the accuracy for a test case. Let the rough surface over the patch be represented by

$$z(x, y) = \sigma \cos(k_x x) \cos(k_y y) \tag{261}$$
$$\sigma = 0.1 \tag{262}$$
$$k_x = \pi/3 \tag{263}$$
$$k_y = \pi/4 \tag{264}$$
$$\tag{265}$$

The surface metric S can be computed from Eq. 30.

$$\frac{\partial z}{\partial x} = -\sigma k_x \sin(k_x x) \cos(k_y y) \tag{266}$$
$$\frac{\partial z}{\partial y} = -\sigma k_y \cos(k_x x) \sin(k_y y) \tag{267}$$
$$S(x, y) = \sqrt{1 + \sigma^2 k_x^2 \sin^2(k_x x) \cos^2(k_y y)^2 + \sigma^2 k_y^2 \cos^2(k_x x) \sin^2(k_y y)^2} \tag{268}$$

6.3. Simple Nyström Example

This is a simplified example for the impedance matrix for a problem with two square patches and one point in the middle of each patch. This has a simple (low order) integration rule. Let the length of the side of each patch be a and the two patches be side by side. The center of the first patch is $(0, 0)$ and the center of the second patch is $(a, 0)$. If the function to be integrated is $f(x, y)$, then the integral of f on the first patch is

$$\int_{-a/2}^{a/2} \int_{-a/2}^{a/2} f(x, y)\, dx\, dy \approx a^2 f(0, 0). \tag{269}$$

The integral on the second patch is

$$\int_{-a/2}^{a/2} \int_{a/2}^{3a/2} f(x,y)\,dx\,dy \approx a^2 f(a,0). \tag{270}$$

Let's let the patch be smooth so that $S(x,y) = 1$ everywhere.

The impedance matrix Z will be a 2×2 matrix. For the off-diagonal matrix elements,

$$Z_{12} = G(0,a)w_2. \tag{271}$$

The Greene's function in this case is

$$G(x,y,x',y') = e^{ik\rho}/\rho \tag{272}$$

where

$$\rho = \sqrt{(x-x')^2} = a. \tag{273}$$

The weighting function is a^2. Therefore

$$G(0,0,a,0) = G(a,0,0,0)) = e^{ika}/a \tag{274}$$

and

$$Z_{12} = Z_{21} = ae^{ika}. \tag{275}$$

The same-patch matrix elements must be exact for a constant impedance matrix (which we obviously don't have). That requires that

$$\int G(x_0,y_0,x',y') = w_i \tag{276}$$

for each patch. (x_0,y_0) is the evaluation point $(0,0)$ for patch 1 and $(a,0)$ for patch 2. The impedance matrix element will then be

$$Z_{ii} = w_i. \tag{277}$$

In this case $w_1 = w_2$. This would not be the case if there was roughness on the surface or if that patches were asymmetric in the evaluation of G on the patch. Here is the Julia code to initialize the HCubature package and define the Greene's function.

```
using HCubature
function G(xp::Float64, yp::Float64)
    k = 2 * pi
    r = sqrt(xp^2+yp^2)
    exp(1im * k * r)/r
end
```

The cubature function works as expected integrating a unit function.

```
a=1.5
hcubature((x)->1,(-a/2,-a/2),(a/2,a/2) )
```

producing the following result.

```
(2.25, 0.0)
```

Using HCubature for a problem with an internal singularity can cause problems. The following code crashed my windows computer.

```
a=1.5
@time hcubature((x)->G(x[1],x[2]),(-a/2,-a/2),(a/2,a/2))
```

Putting the singularity on an end point solved the problem. The integration in this case is a sum of four integrations with the singularity at the corner of each.

```
a=0.1
function qquad()
    a1 = hcubature((x)->G(x[1],x[2]),(-a/2,-a/2),(0.0,0.0))[1]
    a2 = hcubature((x)->G(x[1],x[2]),(-a/2,0.0),(0.0,a/2))[1]
    a3 = hcubature((x)->G(x[1],x[2]),(0.0,0.0),(a/2,a/2))[1]
    a4 = hcubature((x)->G(x[1],x[2]),(0.0,-a/2),(a/2,0.0))[1]
    a1+a2+a3+a4
end
@time qquad()
```

This produced the following result.

```
  2.199127 seconds (4.10 M allocations: 215.024 MiB, 11.79\% gc time)
0.345047995271054 + 0.06214599003394548 3im
```

Note that the adaptive quadrature is not very efficient and uses a lot of memory. A Duffy transformation to remove the singularity would probably been more efficient, but this is okay for my point. This Julia calculation agrees with the following Mathematica calculation.

```
a = 0.1;
k = 2 Pi;
G[xp_, yp_] := Module[{r},
  r = Sqrt[xp^2 + yp^2];
  Exp[I k r]/r
  ]
NIntegrate[G[x, y], {x, -a/2, a/2}, {y, -a/2, a/2},
 Exclusions -> {{0, 0}}]

0.345048 +0.062146 I
```

Thus, if $a = 0.1$ and $k = 2\pi$,

$$Z_{12} = Z_{21} = 0.1e^{0.2\pi i} \tag{278}$$

$$= 0.0809 + 0.0588i \tag{279}$$

$$Z = \begin{pmatrix} 0.345 + 0.0621i & 0.0809 + 0.0588i \\ 0.0809 + 0.0588i & 0.345 + 0.0621i \end{pmatrix} \tag{280}$$

7. Application Notes

Our primary interest in a surface which is not flat is to integrate over a rough surface. Care should be taken in selecting the algorithm for generating the surface so that partial derivatives are easily and accurately evaluated. Two good choices are surfaces characterized by cubic splines and surfaces from band-width limited Fourier transforms.

7.1. Splines

Cubic splines are piece-wise cubic polynomials with continuous first and second partial derivatives. There are two simple ways I can think of to generate such surfaces. One would be to use a Gaussian random number generator to create random surface points which are more widely spaced than the patch separations. The splines will be forced to go through these points (called knots) exactly, but the other points will be interpolated. If the points are relatively far apart, the surface will be smooth with a spatial frequency equal to about 1/3 the spline separation (since the cubic could have three local extrema in general).

Alternatively, the spline could be generated by using a Gaussian random number generator at each grid point and then creating a smoothing spline to interpolate across the region. The smoothing spline has adjustable knots which are varied to minimize the deviation of the spline from the data points. It is could also controlled by a smoothing parameter which is varied to constrain discontinuities in the derivatives in adjoining regions.

The partial derivatives of the splines are easily computed since the interpolation is just a polynomial. Most spline libraries have facilities for computing the partial derivatives for you.

7.2. Fourier Transform

Our AFM data suggests that some surfaces are well-modeled by a noise function which has an envelop like a half-Gaussian in the Fourier domain. Such a surface can be generated by starting with points on the surface with random Gaussian noise added. This surface is then transformed to the frequency domain using a 2d Fast Fourier Transform. In that domain, the Fourier components are multiplied by

$$f(k) = e^{-k^2/2\sigma^2} \tag{281}$$

where σ is a constant chosen to regulate the amount of smoothing and

$$k^2 = k_x^2 + k_y^2 \tag{282}$$

is the wave number (i.e. the independent variable in the Fourier domain. The filtered spectrum is then transformed back to the spatial domain using an inverse fast Fourier transform to generate the required heights.

The partial derivatives are easily computed since the Fourier expansion is just a sum of complex exponentials. One has to take care of high frequency ghosts in the Fourier transforms because of aliasing. I don't think this will be a problem in our case since we are applying a low-pass filter which will suppress them.

A. Sample Julia Code

A.1. Circular Integration

Here is a function applying the numerical circular integration method outlined in Section 2 using Julia.

```
"""
cint integrates the function f over a circle of radius r
using n rings. Each ring is divided into 3(2i+1) segments,
where i=0 for the inner most ring and can have values up to n-1.
"""
function cint(f,r,n)
    a=r/n
    sum=0.0
    for i=0:n-1
        ip=(2*i+1+1/sqrt(3.0))/(2*i+1)
        im=(2*i+1-1/sqrt(3.0))/(2*i+1)
        rp=a*(i+0.5+1/(2*sqrt(3.0)))
        rm=a*(i+0.5-1/(2*sqrt(3.0)))
        for m=0:6*i+2
            tp=2*pi*(m+0.5+1/(2*sqrt(3.0)))/(3*(2*i+1))
            tm=2*pi*(m+0.5-1/(2*sqrt(3.0)))/(3*(2*i+1))
            sum += im*f(rm,tm)+im*f(rm,tp)+ip*f(rp,tm)+ip*(f(rp,tp))
        end
    end
    sum *= pi*a^2/12
end
```

A.2. Same Patch Integration

Even though the Duffy transformation wasn't helpful in the python code (see Sec. B.2), it may be important in Julia because the Cubature package only has contants integration limits. Nested 1d quadratures may work, but the timing could be different. This module tests that.

A.2.1. duffy.jl

```
using Test
using HCubature
using QuadGK
```

```julia
function duffy(func, tol=0)
  hcubature(x->x[1]*func(x[1], x[1]*x[2]),[0.,0.],[1.,1.],atol=tol)
end

@testset "Duffy" begin
  @testset "constant" begin
    area, err = duffy((u,v)->1)
    @test area ≈ 0.5
    @test err < 1e-7
  end
  @testset "singular" begin
    area, err = duffy((u,v)->1/sqrt(u^2+v^2))
    @test area ≈ asinh(1)
    @test err < 1e-7
  end
  @testset "complex" begin
    area, err = duffy((u,v)->exp(1im*pi/4)/sqrt(u^2+v^2))
    @test area ≈ asinh(1)*(1+1im)/sqrt(2)
    @test err < 1e-7
  end
end

println("\nChecking duffy timing")
function or(u,v)
  1/sqrt(u^2+v^2)
end

function df()
  loops = 1000
  a = 0
  for i=1:loops
    a += duffy(or,1e-12)[1]
  end
  a/loops
end

exact = asinh(1)
println("df = $(df()), should be $exact")
@time df()
```

This program has the following output:

```
Test Summary: | Pass  Total
Duffy         |    6      6
```

```
Checking duffy timing
df = 0.8813735870195392, should be 0.881373587019543
0.076966 seconds (1.14 M allocations: 31.342 MiB, 6.80% gc time)
```

A.2.2. surface.jl

```
using Test
using Printf

struct patch
    # For now, I'll sacrifice speed for space. This should only
    # be of order N, and the iminuspedance matrix will be of order N^2
    # for singular patch integrations I need the Jacobian,
end

struct surface
    rings :: Int64  # number of annual rings
    a :: Float64
    patches :: Array{patch,1}
    # The Jacobian and B matrices only depend on a, the triangle numbber,
    # and the singular point number for each patch. Hence it makes sense
    # to store them for the surface and not with each patch.
end

struct triangle
end

# n and m are indexed from 0
# element is indexed from 1
function nm(element::Integer)
    el = element - 1
    n = Int(trunc(sqrt(el/3)))
    m = el - 3*n^2
    (n,m)
end

function elmnt(n::Integer, m::Integer)
    m+3*n^2+1
end

# positive (rp) and negative (rm) radius of annulus number n
# with n starting with zero
```

```
function rplusminus(a::AbstractFloat, n::Integer)
    term = a/(2*sqrt(3.0))
    rmid = a*(n+0.5)
    (rmid, term)
end

# positive (tp) and negative (tm) theta for patch number m
# and annulus n, assuming n and m start with 0
function tplusminus(a::AbstractFloat, n::Integer, m::Integer)
    t = 1/(2*sqrt(3.0))
    d = 3*(2*n+1)
    tdif = 2*pi*t/d
    tmid = 2*pi*(m+0.5)/d
    (tmid, tdif)
end

function dfill(n::Integer, greene, rm::Float64, dr::Float64,
    tm::Float64, dt::Float64, rp::Float64, drp::Float64,
    tp::Float64, dtp::Float64)
    ip = (2*n+1+1/sqrt(3.0))/(2*n+1)
    im = (2*n+1-1/sqrt(3.0))/(2*n+1)
    Z = zeros(4,4)
    Z[1,1] = im*greene(rm-dr, tm-dt, rp-drp, tp-dtp)
    Z[1,2] = im*greene(rm-dr, tm-dt, rp-drp, tp+dtp)
    Z[1,3] = ip*greene(rm-dr, tm-dt, rp+drp, tp+dtp)
    Z[1,4] = ip*greene(rm-dr, tm-dt, rp+drp, tp-dtp)
    Z[2,1] = im*greene(rm-dr, tm+dt, rp-drp, tp-dtp)
    Z[2,2] = im*greene(rm-dr, tm+dt, rp-drp, tp+dtp)
    Z[2,3] = ip*greene(rm-dr, tm+dt, rp+drp, tp+dtp)
    Z[2,4] = ip*greene(rm-dr, tm+dt, rp+drp, tp-dtp)
    Z[3,1] = im*greene(rm+dr, tm+dt, rp-drp, tp-dtp)
    Z[3,2] = im*greene(rm+dr, tm+dt, rp-drp, tp+dtp)
    Z[3,3] = ip*greene(rm+dr, tm+dt, rp+drp, tp+dtp)
    Z[3,4] = ip*greene(rm+dr, tm+dt, rp+drp, tp-dtp)
    Z[4,1] = im*greene(rm+dr, tm-dt, rp-drp, tp-dtp)
    Z[4,2] = im*greene(rm+dr, tm-dt, rp-drp, tp+dtp)
    Z[4,3] = ip*greene(rm+dr, tm-dt, rp+drp, tp+dtp)
    Z[4,4] = ip*greene(rm+dr, tm-dt, rp+drp, tp-dtp)
    Z
end

# Since this function returns Z, it should be called before same_fill
# which fills in the missing same patch parts
function diff_fill(rings::Integer, a::Float64, greene)
```

```
npatch = 3*rings^2
npts = 4*npatch
Z = zeros(npts,npts)
pp = 1 # patch for first index
for np = 0:rings-1
    rmp, drp = rplusminus(a, np)
    for mp = 0:6*np+2
        tmp, dtp = tplusminus(a, np, mp)
        p = 1
        for n = 0:np-1
            rm, dr = rplusminus(a, n)
            for m = 0:6*n+2
                tm, dt = tplusminus(a, n, m)
                Z[p:p+3, pp:pp+3] = dfill(np, greene, rm, dr, tm, dt,
                    rmp, drp,tmp, dtp)*pi*a^2/12
                p += 4
            end # for m
        end # for n
        # n == np (possible same patch)
        rm, dr = rmp, drp
        for m = 0:mp-1
            tm, dt = tplusminus(a, np, m)
            Z[p:p+3, pp:pp+3] = dfill(np, greene, rm, dr, tm, dt,
                rmp, drp, tmp, dtp)*pi*a^2/12
            p += 4
        end # for m
        p += 4 # for the same patch I skipped
        for m = mp+1:6*np+2
            tm, dt = tplusminus(a, np, m)
            Z[p:p+3, pp:pp+3] = dfill(np, greene, rm, dr, tm, dt,
                rmp, drp, tmp, dtp)*pi*a^2/12
            p += 4
        end # for m
        for n = np+1:rings-1
            rm, dr = rplusminus(a, n)
            for m = 0:6*n+2
                tm, dt = tplusminus(a, n, m)
                Z[p:p+3, pp:pp+3] = dfill(np, greene, rm, dr, tm, dt,
                    rmp, drp, tmp, dtp)*pi*a^2/12
                p += 4
            end # for m
        end # for n
        pp = pp+4
    end # for mp
```

```
    end # for np
    Z
end # diff_fill

# This routine fills the Z matrix for same patch elements.
# It is here for testing purposes, since it assumes the greene
# function is not singular. sing_fill is the equivalent routine
# for a singular kernel.
function same_fill(Z::Array{Float64,2}, rings::Integer, a::Float64, greene)
    p = 1 # patch for first index
    for n = 0:rings-1
        rm, dr = rplusminus(a, n)
        for m = 0:6*n+2
            tm, dt = tplusminus(a, n, m)
            Z[p:p+3, p:p+3] = dfill(n, greene, rm, dr, tm, dt,
                            rm, dr, tm, dt)*pi*a^2/12
            p += 4
        end # for m
    end # for n
    Z
end

# testing code
@testset "Z compute" begin
    @testset "indexing" begin
        element = 16
        n, m = nm(element)
        @test n == 2
        @test m == 3
        @test elmnt(n, m) == element
    end
    @testset "points" begin
        # n = 0
        # m = 0
        a = 0.5
        rm, dr = rplusminus(a, 0)
        @test rm ≈ a*0.5
        @test dr ≈ a/(2*sqrt(3.0))
        tm, dt = tplusminus(a, 0, 0)
        @test tm ≈ 2*pi*0.5/3
        @test dt ≈ 2*pi/(2*sqrt(3.0))/3
        # n = 1
        # m = 2
        rm, dr = rplusminus(a, 1)
```

```
    @test rm ≈ a*1.5
    @test dr ≈ a/(2*sqrt(3.0))
    tm, dt = tplusminus(a, 1, 2)
    @test tm ≈ 2*pi*2.5/9
    @test dt ≈ 2*pi/(2*sqrt(3.0))/9
end
@testset "Z different" begin
    rings = 4
    a = 0.5
    Z = diff_fill(rings, a, (r, t, rp, tp) -> 1.5)
    sZ = size(Z)
    # sizes
    @test sZ[1] == 12*rings^2
    @test sZ[2] == sZ[1]
    # first block
    @test Z[1,1] == 0
    @test Z[sZ[1], sZ[1]] == 0
    @test Z[1,5] ≠ 0
    @test Z[25,1] ≠ 0
    # Middle block
    @test Z[25,25] == 0
    @test Z[25,28] == 0
    @test Z[28,25] == 0
    @test Z[25,29] ≠ 0
    @test Z[29,25] ≠ 0
    # Last block
    @test Z[sZ[1],1] ≠ 0
    @test Z[sZ[1],sZ[1]-4] ≠ 0
    @test Z[1,sZ[1]] ≠ 0
    @test Z[sZ[1]-4,sZ[1]] ≠ 0
end
@testset "Z all" begin
    rings = 4
    a = 0.5
    Z = diff_fill(rings, a, (r, t, rp, tp) -> 1.5)
    same_fill(Z, rings, a, (r, t, rp, tp) -> 1.5)
    sZ = size(Z)
    @test Z ≠ zeros(sZ...)
end
@testset "dfill" begin
    rm = 1.0
    dr = 0.25
    rmp = 2.0
    drp = 1/3
```

```
    tm = 3.0
    dt = 0.5
    tmp = 1.5
    dtp = 0.1
    Z = dfill(1, (r, t, rp, tp)->1, rm, dr, tm, dt, rmp, drp, tmp, dtp)
    @test size(Z) == (4,4)
    for j=1:4
        @test Z[:,j] == fill(Z[1,j],4)
    end
    Z = dfill(1, (r, t, rp, tp)->rp, rm, dr, tm, dt, rmp, drp, tmp, dtp)
    for j=1:4
        @test Z[:,j] == fill(Z[1,j],4)
    end
    Z = dfill(1, (r, t, rp, tp)->tp, rm, dr, tm, dt, rmp, drp, tmp, dtp)
    for j=1:4
        @test Z[:,j] == fill(Z[1,j],4)
    end
end
@testset "Constant Int" begin
    rings = 4
    a = 0.1
    Z = diff_fill(rings, a, (r, t, rp, tp) -> 1)
    same_fill(Z, rings, a, (r, t, rp, tp) -> 1)
    sZ = size(Z)[1]
    V = ones(sZ)
    I = Z*V # each element should be the area of the circle
    radius = a*rings
    area = pi*radius^2
    @test area ≈ I[1]
    @test area ≈ I[10]
    @test area ≈ I[sZ]
end
@testset "Linear Int" begin
    rings = 4
    a = 0.1
    Z = diff_fill(rings, a, (r, t, rp, tp) -> rp)
    same_fill(Z, rings, a, (r, t, rp, tp) -> rp)
    sZ = size(Z)[1]
    I = sum(Z,dims=2)
    @test I ≈ fill(I[1],sZ) # all rows should be the same
    radius = a*rings
    exact = 2/3*pi*radius^3
    @test exact ≈ I[1]
    Z = diff_fill(rings, a, (r, t, rp, tp) -> tp)
```

```
        same_fill(Z, rings, a, (r, t, rp, tp) -> tp)
        I = sum(Z, dims=2)
        exact = pi^2*radius^2
        @test I ≈ fill(exact,sZ)
    end
end
```

B. Sample Python Code

This chapter includes a python implementation of these rules with association unit testing. It is a modified version of the routines written by Michael Greenberg.

B.1. Circular Integration

```
# cint.py
# Steve Turley, 10/13/2018

import numpy as np
import unittest

def cint(f, r, n):
    """integrate the function f(r,theta) over a circle of
    radius r using n rings. Each ring is divided into 3(2i+1)
    segments, where i=0 for the innermost ring and can have
    values up to n-1.
    """
    a=r/n
    sum=0.0
    for i in range(n):
        ip=(2*i+1+1/np.sqrt(3))/(2*i+1)
        im=(2*i+1-1/np.sqrt(3))/(2*i+1)
        rp=a*(i+0.5+1/(2*np.sqrt(3)))
        rm=a*(i+0.5-1/(2*np.sqrt(3)))
        for m in range(6*i+3):
            tp=2*np.pi*(m+0.5+1/(2*np.sqrt(3)))/(3*(2*i+1))
            tm=2*np.pi*(m+0.5-1/(2*np.sqrt(3)))/(3*(2*i+1))
            sum += im*(f(rm,tm)+f(rm,tp)) + ip*(f(rp,tm)+f(rp,tp))
    sum *= np.pi*a**2/12
    return sum

class CircInt(unittest.TestCase):

    # test constant integrations
```

```
    def test_const(self):
        self.assertEqual(np.pi,cint(lambda r,t: 1.0,1.0,1))
        self.assertEqual(np.pi,cint(lambda r,t: 1.0,1.0,3))
        self.assertEqual(4*np.pi,cint(lambda r,t:1.0,2.0,2))
        self.assertEqual(12*np.pi,cint(lambda r,t:3.0,2.0,4))

    def test_linr(self):
        self.assertAlmostEqual(2/3*np.pi,cint(
            lambda r,t: r, 1.0, 1), 14)
        self.assertAlmostEqual(2/3*np.pi,cint(
            lambda r,t: r, 1.0, 3), 14)
        self.assertAlmostEqual(16/3*np.pi,cint(
            lambda r,t: r, 2.0, 2), 14)
        self.assertAlmostEqual(2*np.pi,cint(
            lambda r, t: 3*r, 1.0,4), 14)
        self.assertAlmostEqual(np.pi**2,cint(
            lambda r,t: t, 1.0, 1), 14)
        self.assertAlmostEqual(np.pi**2,cint(
            lambda r,t: t, 1.0, 3), 14)
        self.assertAlmostEqual(9*np.pi**2,cint(
            lambda r, t: t, 3.0, 2), 14)
        self.assertAlmostEqual(18*np.pi**2,cint(
            lambda r,t: 2*t, 3.0, 4), 14)

if __name__ == '__main__':
    unittest.main()
```

B.2. Same Patch Integration

This code (patch.py) has the transformations from an arc to the unit right triangle needed for the Duffy transformation. It corresponds to `alt_patch.f95` in Sec. C.2.

```
# patch.py
# Steve Turley, 10/16/2018

import numpy as np
import unittest

class patch:

    def __init__(self, n, m, triangle, singular_point, a):
        self.n = n
        self.m = m
        self.triangle = triangle
```

```
        self.singular_point = singular_point
        self.a = a
        apt = a/(2*np.sqrt(3))
        if singular_point < 3:
            self.xs = -apt
        else:
            self.xs = apt
        if (singular_point == 1) | (singular_point == 4):
            self.ys = -apt
        else:
            self.ys = apt
        pp2 = self.p2()
        pp3 = self.p3()
        qq2 = self.q2()
        qq3 = self.q3()
        self.b= np.array(((pp2, pp3-pp2),(qq2, qq3-qq2)))
        self.j = np.abs(np.linalg.det(self.b))

    def p2(self):
        if self.triangle < 3:
            s1 = -1
        else:
            s1 = 1
        if self.singular_point < 3:
            s2 = -1
        else:
            s2 = 1
        return self.a*(s1/2-s2/(2*np.sqrt(3)))

    def p3(self):
        if (self.triangle == 1) | (self.triangle == 4):
            s1 = -1
        else:
            s1 = 1
        if self.singular_point < 3:
            s2 = -1
        else:
            s2 = 1
        return self.a*(s1/2-s2/(2*np.sqrt(3)))

    def q2(self):
        if (self.triangle == 1) | (self.triangle == 4):
            s1 = -1
        else:
```

```
            s1 = 1
        if (self.singular_point ==1) | (self.singular_point == 4):
            s2 = -1
        else:
            s2 = 1
        return self.a*(s1/2-s2/(2*np.sqrt(3)))

    def q3(self):
        if self.triangle < 3:
            s1 = 1
        else:
            s1 = -1
        if (self.singular_point == 1) | (self.singular_point == 4):
            s2 = -1
        else:
            s2 = 1
        return self.a*(s1/2-s2/(2*np.sqrt(3)))

    def radius(self, xpp, ypp):
        return self.b[1,0]*xpp+self.b[1,1]*ypp+(
                self.ys+self.a*(self.n+0.5))

    def theta(self, xpp, ypp):
        return 2*np.pi*(self.a*(self.m+0.5)+self.b[0,0]*xpp+(
                self.b[0,1]*ypp+self.xs))/(
                        3*self.a*(2*self.n+1))

    def grfunc(self, xpp, ypp):
        k = 2*np.pi
        r = self.radius(xpp, ypp)
        th = self.theta(xpp, ypp)
        rs = self.ys + self.a*(self.n+0.5)
        ths = 2*np.pi*(self.a*(self.m+0.5)+self.xs)/(
                3*self.a*(2*self.n+1))
        rho = np.sqrt(r**2+rs**2-2*r*rs*np.cos(th-ths))
        return np.exp(1j*k*rho)/(4*np.pi*rho)

    def jacobian(self):
        return self.j*2*np.pi/(self.a*(6*self.n+3))

class PatchTest(unittest.TestCase):
    def test_init(self):
        n = 2
        m = 7
```

```
        t = 1
        sp = 2
        a = 0.5
        p = patch(n, m, t, sp, a)
        self.assertEqual(n, p.n)
        self.assertEqual(m, p.m)
        self.assertEqual(t, p.triangle)
        self.assertEqual(sp, p.singular_point)
        apt = 0.1443376
        self.assertAlmostEqual(-apt, p.xs, 7)
        self.assertAlmostEqual(apt, p.ys, 7)
        t = 2
        sp = 4
        p = patch(n, m, t, sp, a)
        self.assertAlmostEqual(apt, p.xs, 7)
        self.assertAlmostEqual(-apt, p.ys, 7)


    def test_bmat(self):
        p = patch(2, 7, 1, 2, 0.5)
        # comparing with FORTRAN answers
        c1 = 0.1056624
        c2 = 0.3943376
        self.assertAlmostEqual(-c1, p.b[0,0], 7)
        self.assertAlmostEqual(0, p.b[0,1], 7)
        self.assertAlmostEqual(-c2, p.b[1,0], 7)
        self.assertAlmostEqual(0.5, p.b[1,1], 7)
        self.assertAlmostEqual(0.0528312, p.j, 6)
        # try another
        p = patch(2, 7, 2, 4, 0.5)
        self.assertAlmostEqual(-c2, p.b[0,0], 7)
        self.assertAlmostEqual(0.5, p.b[0,1], 7)
        self.assertAlmostEqual(c2, p.b[1,0], 7)
        self.assertAlmostEqual(0.0, p.b[1,1], 7)
        self.assertAlmostEqual(0.1971688, p.j, 7)

    def test_func(self):
        p = patch(2,7,1,2,0.5)
        # comparing with FORTRAN answers
        xp = 0.5
        yp = 0.5
        self.assertAlmostEqual(1.4471688, p.radius(xp, yp), 7)
        self.assertAlmostEqual(2.9764129, p.theta(xp, yp), 7)
        self.assertAlmostEqual(0.8429136+0.4781089j,
```

```
                        p.grfunc(xp, yp), 7)
        self.assertAlmostEqual(0.0442598, p.jacobian(), 6)
        # try another triangle and singular point
        p = patch(2, 7, 2, 4, 0.5)
        self.assertAlmostEqual(1.3028312, p.radius(xp, yp), 7)
        self.assertAlmostEqual(3.3067724, p.theta(xp, yp), 7)
        self.assertAlmostEqual(0.1106089+0.3736807j,
                        p.grfunc(xp, yp), 6)
        self.assertAlmostEqual(0.1651797, p.jacobian(), 7)

if __name__ == '__main__':
    unittest.main()
```

This code (duffy.py) has the code and testing information for applying the duffy transformation for doing a singular integral with the singularily at the corner of a rigth triangle (0,0). When I did a timing test, I found that the duffy integration took a little longer than just doing an adaptive double integral using `scipy.integrate.dblquad`.

```
import numpy as np
import scipy.integrate as int
import unittest

def duffy_int(func, tol):
    return int.dblquad(lambda v,u : u*func(u,u*v), 0, 1,
                lambda x : 0, lambda x : 1, epsabs = tol)

# This is inefficient. You will often co better breaking the
# integrand up into real and imaginary parts beforehand
def duffy_qint(func, tol):
    r, e = int.dblquad(lambda v,u : u*np.real(func(u,u*v)), 0, 1,
                lambda x : 0, lambda x : 1, epsabs = tol)
    i, e = int.dblquad(lambda v,u : u*np.imag(func(u,u*v)), 0, 1,
                lambda x : 0, lambda x : 1, epsabs = tol)
    return r+i*1j

class DuffyTest(unittest.TestCase):
    def test_spint(self):
        # Integrate half circle as a test with
        # x going from -2 to 2 and y going from g(x)=-0
        # to h(x)=sqrt(4-x^2)
        area, error = int.dblquad(lambda y, x : 1.0, -2.0, 2.0,
                                lambda x : 0.0,
                                lambda x : np.sqrt(4.0-x**2),
                                epsabs = 1e-12)
        self.assertAlmostEqual(2*np.pi, area, 12)
```

```
        self.assertAlmostEqual(0.0, error, 7)
        # duffy integration
        area, error = int.dblquad(lambda y, x : 1, 0., 1.,
                                  lambda x : 0.,
                                  lambda x : x, epsabs = 1e-12)
        self.assertAlmostEqual(0.5, area, 12)

    def test_duffy(self):
        # constant integral of unit right triangle
        area, error = duffy_int(lambda u, v: 1, 1e-12)
        self.assertAlmostEqual(0.5, area, 12)
        self.assertAlmostEqual(0.0, error, 12)
        dbarea, error = int.dblquad(lambda y, x :
            1/np.sqrt(x**2+y**2), 0, 1,lambda x: 0,
            lambda x : x ,epsrel = 1e-12)
        self.assertAlmostEqual(0, error, 12)
        area, error = duffy_int(lambda u, v:
            1/np.sqrt(u**2+v**2), 1e-12)
        self.assertAlmostEqual(dbarea, area, 15)

    def test_cmplx(self):
        # try a complex integrand
        cnst = (1+1j)/np.sqrt(2)
        area = duffy_qint(lambda y, x : cnst, 1e-12)
        self.assertAlmostEqual(cnst*0.5, area, 12)

if __name__ == '__main__':
    unittest.main()
```

C. Sample FORTRAN Code

This section has two FORTRAN implementations for computing same patch integrals with singular kernels: `patch.f95` and `alt_patch.f95`. It also includes the unit testing code `test_patch.pf` based on the pFUnit testing framework.

C.1. patch.f95

```
! Routines and constants specialized to integration over a particular
! patch and triangle.
! Steve Turley, August 8, 2017
module patch
  use iso_fortran_env, only : real64
  implicit none
```

```
  private
  type patch_par
     ! private
     ! public
     integer :: n, m, triangle, singular_point
     real(real64) :: a
     real(real64) :: b(2,2), j
     real(real64) :: xs, ys
   contains
     procedure :: radius
     procedure :: theta
     procedure :: grfunc ! Greene's function
     procedure :: jacobian
  end type patch_par

  ! constructor(s)
  interface patch_par
     module procedure :: init
  end interface patch_par

  public :: patch_par

contains

  function init(n,m,triangle, singular_point, a)
    integer, intent(in) :: n, m, triangle, singular_point
    real(real64), intent(in) :: a
    type(patch_par) init
    real(real64) :: apt
    init%n = n
    init%m = m
    init%triangle = triangle
    init%a = a
    init%singular_point = singular_point
   ! Set singular points xs and ys
    apt = a/(2*sqrt(3.0d0))
    if(singular_point < 3) then
       init%xs = -apt
    else
       init%xs = apt
    end if
    if(singular_point == 1 .OR. singular_point == 4) then
       init%ys = -apt
    else
```

```fortran
    init%ys = apt
  end if
  ! This is where the B's and J's are initialized. It isn't final yet
  select case (10*singular_point+triangle)
  case(11)
    init%b = reshape((/ -a*(3-sqrt(3.d0))/6, -a*(3-sqrt(3.d0))/6,&
          0.0d0, a /), shape(init%b))
    init%j = a**2*(3-sqrt(3.d0))/6
  case(21)
    init%b = reshape((/ -a*(3-sqrt(3.d0))/6, -a*(3+sqrt(3.d0))/6,&
          0.0d0, a /), shape(init%b))
    init%j = a**2*(3-sqrt(3.d0))/6
  case(31)
    init%b = reshape((/ -a*(3+sqrt(3.d0))/6, -a*(3+sqrt(3.d0))/6,&
          0.0d0, a /), shape(init%b))
    init%j = a**2*(3+sqrt(3.d0))/6
  case(41)
    init%b = reshape((/ -a*(3+sqrt(3.d0))/6, -a*(3-sqrt(3.d0))/6,&
          0.0d0, a /), shape(init%b))
    init%j = a**2*(3+sqrt(3.d0))/6
  case(12)
    init%b = reshape((/ -a*(3-sqrt(3.d0))/6, a*(3+sqrt(3.d0))/6,&
          a, 0.d0 /), shape(init%b))
    init%j = a**2*(3+sqrt(3.d0))/6
  case(22)
    init%b = reshape((/ -a*(3-sqrt(3.d0))/6, a*(3-sqrt(3.d0))/6,&
          a, 0.d0 /), shape(init%b))
    init%j = a**2*(3-sqrt(3.d0))/6
  case(32)
    init%b = reshape((/ -a*(3+sqrt(3.d0))/6, a*(3-sqrt(3.d0))/6,&
          a, 0.d0 /), shape(init%b))
    init%j = a**2*(3-sqrt(3.d0))/6
  case(42)
    init%b = reshape((/ -a*(3+sqrt(3.d0))/6, a*(3+sqrt(3.d0))/6,&
          a, 0.d0 /), shape(init%b))
    init%j = a**2*(3+sqrt(3.d0))/6
  case(13)
    init%b = reshape((/ a*(3+sqrt(3.d0))/6, a*(3+sqrt(3.d0))/6,&
          0.0d0, -a /), shape(init%b))
    init%j = a**2*(3+sqrt(3.d0))/6
  case(23)
    init%b = reshape((/ a*(3+sqrt(3.d0))/6, a*(3-sqrt(3.d0))/6,&
          0.0d0, -a /), shape(init%b))
    init%j = a**2*(3+sqrt(3.d0))/6
```

```
    case(33)
      init%b = reshape((/ a*(3-sqrt(3.d0))/6, a*(3-sqrt(3.d0))/6,&
           0.0d0, -a /), shape(init%b))
      init%j = a**2*(3-sqrt(3.d0))/6
    case(43)
      init%b = reshape((/ a*(3-sqrt(3.d0))/6, a*(3+sqrt(3.d0))/6,&
           0.0d0, -a /), shape(init%b))
      init%j = a**2*(3-sqrt(3.d0))/6
    case(14)
      init%b = reshape((/ a*(3+sqrt(3.d0))/6, -a*(3-sqrt(3.d0))/6,&
           -a, 0.d0 /), shape(init%b))
      init%j = a**2*(3-sqrt(3.d0))/6
    case(24)
      init%b = reshape((/ a*(3+sqrt(3.d0))/6, -a*(3+sqrt(3.d0))/6,&
           -a, 0.d0 /), shape(init%b))
      init%j = a**2*(3+sqrt(3.d0))/6
    case(34)
      init%b = reshape((/ a*(3-sqrt(3.d0))/6, -a*(3+sqrt(3.d0))/6,&
           -a, 0.d0 /), shape(init%b))
      init%j = a**2*(3+sqrt(3.d0))/6
    case(44)
      init%b = reshape((/ a*(3-sqrt(3.d0))/6, -a*(3-sqrt(3.d0))/6,&
           -a, 0.d0 /), shape(init%b))
      init%j = a**2*(3-sqrt(3.d0))/6
    end select
end function init

  function radius(this, xpp, ypp)
    class(patch_par) this
    real(real64), intent(in) :: xpp, ypp
    real(real64)::radius
    radius = this%b(2,1)*xpp+this%b(2,2)*ypp+this%ys+this%a&
         *(this%n+0.5d0)
  end function radius

  function theta(this, xpp, ypp)
    class(patch_par) this
    real(real64), intent(in) :: xpp, ypp
    real(real64)::theta
    real(real64), parameter :: pi = 4.0d0*atan(1.0d0)
    theta = 2*pi*(this%a*(this%n+0.5d0)+this%b(1,1)*xpp+&
         this%b(1,2)*ypp+this%xs)/(3*this%a*(2*this%n+1))
  end function theta
```

```
  function grfunc(this, xpp, ypp)
    class(patch_par) this
    real(real64), intent(in) :: xpp, ypp
    complex(real64)::grfunc
    real(real64), parameter :: pi = 4.0d0*atan(1.0d0)
    real(real64), parameter :: k = 2*pi
    real(real64) :: rho, r, th, rs, ths
    r = this%radius(xpp, ypp)
    th = this%theta(xpp, ypp)
    rs = this%ys + this%a*(this%n+0.5d0)
    ths = 2*pi*(this%a*(this%m+0.5d0)+this%xs)/(3*this%a*(2*this%n+1))
    rho = sqrt(r**2+rs**2-2*r*rs*cos(th-ths))
    grfunc = exp(cmplx(0.d0,k*rho,real64)) /(4*pi*rho)
  end function grfunc

  function jacobian(this)
    class(patch_par) this
    real(real64)::jacobian
    real(real64), parameter :: pi = 4.0d0*atan(1.0d0)
    jacobian = this%j*2*pi/(this%a*(6*this%n+3))
  end function jacobian

end module patch
```

C.2. alt_patch.f95

```
! Routines and constants specialized to integration over a particular
! patch and triangle.
! Steve Turley, August 10, 2017
!
! This is an alternative to the patch module which changes the numbering
! of the singular points and uses an algorithmic approach to setting
! the b and j variables. The b's and j's are different than those computed
! by the patch module.
!
module alt_patch
  use iso_fortran_env, only : real64
  implicit none
  private
  type patch_par
     ! private
     ! public
     integer :: n, m, triangle, singular_point
```

```
    real(real64) :: a
    real(real64) :: b(2,2), j
    real(real64) :: xs, ys
  contains
    procedure :: radius
    procedure :: theta
    procedure :: grfunc ! Greene's function
    procedure :: jacobian
 end type patch_par

 ! constructor(s)
 interface patch_par
    module procedure :: init
 end interface patch_par

 public :: patch_par

contains

 function init(n,m,triangle, singular_point, a)
   integer, intent(in) :: n, m, triangle, singular_point
   real(real64), intent(in) :: a
   type(patch_par) init
   real(real64) :: apt, pp2, qq2, pp3, qq3
   init%n = n
   init%m = m
   init%triangle = triangle
   init%a = a
   init%singular_point = singular_point
   ! Set singular points xs and ys
   apt = a/(2*sqrt(3.0d0))
   if(singular_point < 3) then
      init%xs = -apt
   else
      init%xs = apt
   end if
   if(singular_point == 1 .OR. singular_point == 4) then
      init%ys = -apt
   else
      init%ys = apt
   end if
  ! This is where the B's and J's are initialized.
   pp2 = p2(a, triangle,singular_point)
   pp3 = p3(a, triangle,singular_point)
```

```
  qq2 = q2(a, triangle,singular_point)
  qq3 = q3(a, triangle,singular_point)
  init%b=reshape([pp2,qq2,pp3-pp2,qq3-qq2],shape(init%b))
  init%j=adet(init%b)
end function init

function p2(a, t,s)
  real(real64), intent(in) :: a
  integer, intent(in) :: t,s
  real(real64) :: p2
  integer :: s1, s2
  if(t<3) then
     s1=-1
  else
     s1=1
  end if
  if(s<3) then
     s2=-1
  else
     s2=1
  end if
  p2=a*(s1/2.0d0-s2/(2*sqrt(3.d0)))
end function p2

function q2(a, t, s)
  real(real64), intent(in) :: a
  integer, intent(in) :: t,s
  real(real64) :: q2
  integer :: s1, s2
  if(t==1 .OR. t==4) then
     s1=-1
  else
     s1=1
  end if
  if(s==1 .OR. s==4) then
     s2=-1
  else
     s2=1
  end if
  q2=a*(s1/2.0d0-s2/(2*sqrt(3.d0)))
end function q2

function p3(a, t, s)
  real(real64), intent(in) :: a
```

```
    integer, intent(in) :: t,s
    real(real64) :: p3
    integer :: s1, s2
    if(t==1 .OR. t==4) then
       s1=-1
    else
       s1=1
    end if
    if(s<3) then
       s2=-1
    else
       s2=1
    end if
    p3=a*(s1/2.0d0-s2/(2*sqrt(3.d0)))
  end function p3

  function q3(a, t,s)
    real(real64), intent(in) :: a
    integer, intent(in) :: t,s
    real(real64) :: q3
    integer :: s1, s2
    if(t<3) then
       s1=1
    else
       s1=-1
    end if
    if(s==1 .OR. s==4) then
       s2=-1
    else
       s2=1
    end if
    q3=a*(s1/2.0d0-s2/(2*sqrt(3.d0)))
  end function q3

  function adet(b)
    real(real64), intent(in) :: b(2,2)
    real(real64) :: adet
    adet = abs(b(1,1)*b(2,2)-b(1,2)*b(2,1))
  end function adet

  function radius(this, xpp, ypp)
    class(patch_par) this
    real(real64), intent(in) :: xpp, ypp
    real(real64)::radius
```

```
    radius = this%b(2,1)*xpp+this%b(2,2)*ypp+this%ys+this%a&
         *(this%n+0.5d0)
  end function radius

  function theta(this, xpp, ypp)
    class(patch_par) this
    real(real64), intent(in) :: xpp, ypp
    real(real64)::theta
    real(real64), parameter :: pi = 4.0d0*atan(1.0d0)
    theta = 2*pi*(this%a*(this%m+0.5d0)+this%b(1,1)*xpp+&
         this%b(1,2)*ypp+this%xs)/(3*this%a*(2*this%n+1))
  end function theta

  function grfunc(this, xpp, ypp)
    class(patch_par) this
    real(real64), intent(in) :: xpp, ypp
    complex(real64)::grfunc
    real(real64), parameter :: pi = 4.0d0*atan(1.0d0)
    real(real64), parameter :: k = 2*pi
    real(real64) :: rho, r, th, rs, ths
    r = this%radius(xpp, ypp)
    th = this%theta(xpp, ypp)
    rs = this%ys + this%a*(this%n+0.5d0)
    ths = 2*pi*(this%a*(this%m+0.5d0)+this%xs)/(3*this%a*(2*this%n+1))
    rho = sqrt(r**2+rs**2-2*r*rs*cos(th-ths))
    grfunc = exp(cmplx(0.d0,k*rho,real64)) /(4*pi*rho)
  end function grfunc

  function jacobian(this)
    class(patch_par) this
    real(real64)::jacobian
    real(real64), parameter :: pi = 4.0d0*atan(1.0d0)
    jacobian = this%j*2*pi/(this%a*(6*this%n+3))
  end function jacobian

end module alt_patch
```

C.3. test_patch.pf

This code is for unit testing the path and alt_patch modules using pFUnit.

```
@test
subroutine Parameters
  ! Test the patch parameters
```

```
  use iso_fortran_env
  ! use patch
  use alt_patch
  use pfunit_mod
  implicit none
  integer, parameter :: n=3, m=2, triangle=1, singular_point=2
  real(real64), parameter :: a=0.5d0, xpp=0.5d0, ypp=0.5d0
  type(patch_par) pp
  real(real64) :: xs, ys, apt
  integer :: sp
  pp = patch_par(n, m, triangle, singular_point, a)
  @assertEqual(n, pp%n)
  @assertEqual(m, pp%m)
  @assertEqual(a, pp%a,1d-15)
  apt = a/(2*sqrt(3.0d0))
  do sp=1,4
     pp = patch_par(n,m,triangle, sp, a)
     select case(sp)
     case(1)
        xs=-apt
        ys=-apt
     case(2)
        xs=-apt
        ys=apt
     case(3)
        xs=apt
        ys=apt
     case(4)
        xs=apt
        ys=-apt
     end select
     @assertEqual(xs,pp%xs, 1d-14)
     @assertEqual(ys,pp%ys, 1d-14)
  end do
end subroutine Parameters

@test
subroutine altB
  ! Test the patch parameters
  use iso_fortran_env
  use alt_patch
  use pfunit_mod
  implicit none
  integer, parameter :: n=3, m=2
```

```
real(real64), parameter :: a=0.5d0
type(patch_par) pp
real(real64), parameter :: bm = a*(3-sqrt(3.0d0))/6
real(real64), parameter :: bp = a*(3+sqrt(3.0d0))/6
! Check B
pp = patch_par(n, m, 1, 1, a)
@assertEqual(-bm, pp%b(1,1),1d-14)
@assertEqual(-bm, pp%b(2,1),1d-14)
@assertEqual(a,pp%b(2,2),1d-14)
@assertEqual(0.d0,pp%b(1,2),1d-14)
pp = patch_par(n,m,1,2,a)
@assertEqual(-bm, pp%b(1,1),1d-14)
@assertEqual(0.d0, pp%b(1,2), 1d-14)
@assertEqual(-bp, pp%b(2,1), 1d-14)
@assertEqual(a, pp%b(2,2), 1d-14)
pp = patch_par(n, m, 1, 3, a)
@assertEqual(-bp, pp%b(1,1), 1d-14)
@assertEqual(0.d0, pp%b(1,2), 1d-14)
@assertEqual(-bp, pp%b(2,1), 1d-14)
@assertEqual(a, pp%b(2,2), 1d-14)
pp = patch_par(n, m, 1, 4, a)
@assertEqual(-bp, pp%b(1,1), 1d-14)
@assertEqual(0.d0, pp%b(1,2), 1d-14)
@assertEqual(-bm,pp%b(2,1), 1d-14)
@assertEqual(a, pp%b(2,2), 1d-14)
pp = patch_par(n, m, 2, 1, a)
@assertEqual(-bm, pp%b(1,1), 1d-14)
@assertEqual(a, pp%b(1,2),1d-14)
@assertEqual(bp, pp%b(2,1), 1d-14)
@assertEqual(0.d0,pp%b(2,2), 1d-14)
pp = patch_par(n, m, 2, 2, a)
@assertEqual(-bm, pp%b(1,1), 1d-14)
@assertEqual(a, pp%b(1,2), 1d-14)
@assertEqual(bm, pp%b(2,1), 1d-14)
@assertEqual(0.d0, pp%b(2,2), 1d-14)
pp = patch_par(n, m, 2, 3, a)
@assertEqual(-bp, pp%b(1,1), 1d-14)
@assertEqual(a, pp%b(1,2), 1d-14)
@assertEqual(bm, pp%b(2,1), 1d-14)
@assertEqual(0.d0, pp%b(2,2), 1d-14)
pp = patch_par(n, m, 2, 4, a)
@assertEqual(-bp, pp%b(1,1), 1d-14)
@assertEqual(a, pp%b(1,2), 1d-14)
@assertEqual(bp, pp%b(2,1), 1d-14)
```

```
    @assertEqual(0.d0, pp%b(2,2), 1d-14)
    pp = patch_par(n, m, 3, 1, a)
    @assertEqual(bp, pp%b(1,1), 1d-14)
    @assertEqual(0.d0, pp%b(1,2), 1d-14)
    @assertEqual(bp, pp%b(2,1), 1d-14)
    @assertEqual(-a, pp%b(2,2), 1d-14)
    pp = patch_par(n, m, 3, 2, a)
    @assertEqual(bp, pp%b(1,1), 1d-14)
    @assertEqual(0.d0, pp%b(1,2), 1d-14)
    @assertEqual(bm, pp%b(2,1), 1d-14)
    @assertEqual(-a, pp%b(2,2), 1d-14)
    pp = patch_par(n, m, 3, 3, a)
    @assertEqual(bm, pp%b(1,1), 1d-14)
    @assertEqual(0.d0, pp%b(1,2), 1d-14)
    @assertEqual(bm,pp%b(2,1), 1d-14)
    @assertEqual(-a, pp%b(2,2), 1d-14)
    pp = patch_par(n, m, 3, 4, a)
    @assertEqual(bm, pp%b(1,1), 1d-14)
    @assertEqual(0.d0, pp%b(1,2), 1d-14)
    @assertEqual(bp, pp%b(2,1), 1d-14)
    @assertEqual(-a, pp%b(2,2), 1d-14)
    pp = patch_par(n, m, 4, 1, a)
    @assertEqual(bp, pp%b(1,1), 1d-14)
    @assertEqual(-a, pp%b(1,2), 1d-14)
    @assertEqual(-bm, pp%b(2,1), 1d-14)
    @assertEqual(0.d0, pp%b(2,2), 1d-14)
    pp = patch_par(n, m, 4, 2, a)
    @assertEqual(bp, pp%b(1,1), 1d-14)
    @assertEqual(-a, pp%b(1,2), 1d-14)
    @assertEqual(-bp, pp%b(2,1), 1d-14)
    @assertEqual(0.d0, pp%b(2,2), 1d-14)
    pp = patch_par(n, m, 4, 3, a)
    @assertEqual(bm, pp%b(1,1), 1d-14)
    @assertEqual(-a, pp%b(1,2), 1d-14)
    @assertEqual(-bp, pp%b(2,1), 1d-14)
    @assertEqual(0.d0, pp%b(2,2), 1d-14)
    pp = patch_par(n,m,4,4,a)
    @assertEqual(bm, pp%b(1,1), 1d-14)
    @assertEqual(-a, pp%b(1,2), 1d-14)
    @assertEqual(-bm, pp%b(2,1), 1d-14)
    @assertEqual(0.d0, pp%b(2,2), 1d-14)
end subroutine altB

@test
```

```fortran
subroutine altJ
  ! Test the patch parameters
  use iso_fortran_env
  use alt_patch
  use pfunit_mod
  implicit none
  integer, parameter :: n=3, m=2
  real(real64), parameter :: a=0.5d0
  type(patch_par) pp
  integer :: sp, tr
  real(real64), parameter :: bm = a*(3-sqrt(3.0d0))/6
  real(real64), parameter :: bp = a*(3+sqrt(3.0d0))/6
  real(real64), parameter :: jm = bm*a
  real(real64), parameter :: jp = bp*a
  character(80) :: msg
  ! Check J
  pp = patch_par(n, m, 1, 1, a)
  @assertEqual(jm, pp%j,1d-14)
  pp = patch_par(n,m,1,2,a)
  @assertEqual(jm, pp%j, 1d-14)
  pp = patch_par(n, m, 1, 3, a)
  @assertEqual(jp, pp%j, 1d-14)
  pp = patch_par(n, m, 1, 4, a)
  @assertEqual(jp, pp%j, 1d-14)
  pp = patch_par(n, m, 2, 1, a)
  @assertEqual(jp, pp%j, 1d-14)
  pp = patch_par(n, m, 2, 2, a)
  @assertEqual(jm, pp%j, 1d-14)
  pp = patch_par(n, m, 2, 3, a)
  @assertEqual(jm, pp%j, 1d-14)
  pp = patch_par(n, m, 2, 4, a)
  @assertEqual(jp, pp%j, 1d-14)
  pp = patch_par(n, m, 3, 1, a)
  @assertEqual(jp, pp%j, 1d-14)
  pp = patch_par(n, m, 3, 2, a)
  @assertEqual(jp, pp%j, 1d-14)
  pp = patch_par(n, m, 3, 3, a)
  @assertEqual(jm, pp%j, 1d-14)
  pp = patch_par(n, m, 3, 4, a)
  @assertEqual(jm, pp%j, 1d-14)
  pp = patch_par(n, m, 4, 1, a)
  @assertEqual(jm, pp%j, 1d-14)
  pp = patch_par(n, m, 4, 2, a)
  @assertEqual(jp, pp%j, 1d-14)
```

```
  pp = patch_par(n, m, 4, 3, a)
  @assertEqual(jp, pp%j, 1d-14)
  pp = patch_par(n,m,4,4,a)
  @assertEqual(jm, pp%j, 1d-14)
  do tr=1,4
     do sp=1,4
        pp = patch_par(n,m,tr,sp,a)
        write(msg, fmt='("for triangle ",i1," and s.p. ",i1)')tr,sp
        @assertEqual(adet(pp%b), pp%j,1d-14,msg)
     end do
  end do
contains
  function adet(b)
    use iso_fortran_env, only: real64
    implicit none
    real(real64), intent(in) :: b(2,2)
    real(real64) :: adet
    adet=abs(b(1,1)*b(2,2)-b(1,2)*b(2,1))
  end function adet
end subroutine altJ

@test
subroutine patchPar
  ! Test the patch parameters
  use iso_fortran_env
  use alt_patch, only : apatch=>patch_par
  use patch, only: ppatch=>patch_par
  use pfunit_mod
  implicit none
  real(real64), parameter :: a=0.5d0
  type(ppatch) pp
  type(apatch) ap
  integer, parameter :: n=3, m=2, triangle=1, singular_point=2
  integer :: sp, tr

  pp = ppatch(n, m, triangle, singular_point, a)
  ap = apatch(n, m, triangle, singular_point, a)
  @assertEqual(ap%n, pp%n)
  @assertEqual(ap%m, pp%m)
  @assertEqual(ap%a, pp%a,1d-15)
  do tr=1,4
     do sp=1,4
        pp = ppatch(n, m, tr, sp, a)
        ap = apatch(n, m, tr, sp, a)
```

```
        @assertEqual(ap%xs,pp%xs, 1d-14)
        @assertEqual(ap%ys,pp%ys, 1d-14)
     end do
  end do
end subroutine patchPar

@test
subroutine patchBJ
  ! Test the patch parameters
  use iso_fortran_env
  use alt_patch, only : apatch=>patch_par
  use patch, only: ppatch=>patch_par
  use pfunit_mod
  implicit none
  real(real64), parameter :: a=0.5d0
  type(ppatch) pp
  type(apatch) ap
  integer, parameter :: n=3, m=2
  integer :: sp, tr, bx, by
  character(80) :: msg

  do tr=1,4
     do sp=1,4
        pp = ppatch(n,m,tr,sp,a)
        ap = apatch(n,m,tr,sp,a)
        write(msg, fmt='("J for triangle ",i1," and s.p. ",i1)')tr,sp
        @assertEqual(ap%j, pp%j, 1d-14, trim(msg))
        do bx=1,2
           do by=1,2
              write(msg, fmt='("b for tr",i1," sp",i1," bx",i1," by",i1)')tr,s
              @assertEqual(ap%b(bx,by),pp%b(bx,by), 1d-14, trim(msg))
           end do
        end do
     end do
  end do
end subroutine patchBJ

@test
subroutine constInt
  use iso_fortran_env, only : real64
  use alt_patch
  use duffy
  use pfunit_mod
  implicit none
```

```
  integer, parameter :: n=3, m=2, triangle=1, singular_point=2
  real(real64), parameter :: a=0.50, xpp=0.5d0, ypp=0.5d0
  real(real64) :: rad, the, jac, rint
  complex(real64) :: grf
  real(real64), parameter :: mradius = 1.94716878364870d0 ! from Mathematica
  real(real64), parameter :: mtheta=0.630012726620808d0 ! from Mathematica
  complex(real64), parameter :: mgreene=cmplx(0.865006682017783d0,0.4789609952
  real(real64), parameter :: mjacobian=0.0316141259370375d0 ! Mathematica
  real(real64), parameter :: apt = a/(2*sqrt(3.d0))
  real(real64), parameter :: mconst = 0.261799d0
  type(patch_par) pp
  integer :: t

  pp = patch_par(n, m, triangle, singular_point, a)
  rad = pp%radius(xpp, ypp)
  @assertEqual(mradius,rad,1d-14,"radius")
  the = pp%theta(xpp, ypp)
  @assertEqual(mtheta,the,1d-14,"theta")
  @assertEqual(-apt,pp%xs,1d-14,"xs")
  @assertEqual(apt,pp%ys,1d-14,"ys")
  grf = pp%grfunc(xpp, ypp)
  ! interesting that the next test requires this loose of a tolerance
  @assertEqual(mgreene,grf,1d-13)
  jac = pp%jacobian()
  @assertEqual(mjacobian,jac,1d-14,"jacobian")

  ! Diagnostics using simple constant integral
  rint = 0.d0
  do t=1,4
     pp = patch_par(n, m, t, singular_point, a)
     rint = rint+duffy_int(const, 1d-10)*pp%jacobian()
  end do
  @assertEqual(mconst,rint,1d-6,"constant integral")

contains

  function const(x,y)
    real(real64), intent(in) :: x,y
    real(real64) :: const
    const = pp%radius(x,y)
  end function const

end subroutine constInt
```

```
@test
subroutine GreeneInt
  use iso_fortran_env, only : real64
  use alt_patch
  use duffy
  use pfunit_mod
  implicit none
  integer, parameter :: n=3, m=2, singular_point = 2
  real(real64), parameter :: a=0.50, xpp=0.5d0, ypp=0.5d0
  complex(real64) :: gint
  complex(real64), parameter :: mint=cmplx(0.0489868, 0.0731842, real64)
  real(real64), parameter :: apt = a/(2*sqrt(3.d0))
  type(patch_par) pp
  integer :: t

  gint = 0.d0
  do t=1,4
     pp = patch_par(n, m, t, singular_point, a)
     gint = gint + duffy_int(cdf, 1d-10)*pp%jacobian()
  end do
  @assertEqual(mint, gint, 1d-6, "constant with Greene function integral")
contains

  function cdf(x,y)
    real(real64), intent(in) :: x,y
    complex(real64) :: cdf
    cdf = pp%radius(x,y)*pp%grfunc(x,y)
  end function cdf

end subroutine GreeneInt
```

References

[1] National Institute of Standards and Technology, Digital Library of Mathematical Functions (http://dlmf.nist.gov/3.5#x, accessed June 21, 2017).