Assignment #1

January 24, 2019

1 Phys 581 Winter 2019

2 Assignment #1: Error Function

2.1 Alexander Hickey, 10169582

```
In [1]: #Import useful libraries
        import numpy as np
        import time
        import scipy.special as special
        import matplotlib.pyplot as plt
        import matplotlib.colors as colors
        %matplotlib inline

        #My library
        import erftools as mylib
```

2.1.1 Introduction

The error function is a special function that shows up frequently in probability theory and partial differential equations that describe diffusion. It is defined as:

$$\mathrm{erf}(x) := \frac{1}{\sqrt{\pi}} \int_{-x}^{x} e^{-t^2}\, dt = \frac{2}{\sqrt{\pi}} \int_{0}^{x} e^{-t^2}\, dt.$$

For nonnegative values of x, the error function has the interpretation of describing the area under a Gaussian distribution. In particular, for a random variable Y that is normally distributed with mean 0 and variance $1/2$, $\mathrm{erf}(x)$ describes the probability of Y falling in the interval $[x, x]$. A related function that is often encountered is the Faddeeva function, defined as

$$w(z) = e^{-z^2}\left(1 - \mathrm{erf}(-iz)\right)$$

Being a special function, the error function has no closed form expression as elementary functions. It is however, an entire function; it has no singularities (except that at infinity) and so its Taylor series converges everywhere. Expanding the integrand e^{z^2} into its Maclaurin series and integrating term by term, one obtains the error function's Maclaurin series

$$\mathrm{erf}(z) = \frac{2}{\sqrt{\pi}} \sum_{n=0}^{\infty} \frac{(-1)^n z^{2n+1}}{n!(2n+1)} = \frac{2}{\sqrt{\pi}} \left(z - \frac{z^3}{3} + \frac{z^5}{10} - \frac{z^7}{42} + \frac{z^9}{216} - \cdots \right)$$

which holds for every complex number z. This expansion provides a simple way to compute values of the error function numerically. Over the years, there has been a significant effort to simplify the computation of the error function and improve the numerical stability.

In this notebook, we will examine various approaches to computing values of the complex error function. We will start by comparing a direct computation of the Taylor series to the function available in the SciPy distribution: special.erf. We will also implement a more complex algorithm, known as the Gautschi algorithm (https://dl.acm.org/citation.cfm?id=363618), which turns out to be more numerically stable than the Taylor series approach for large $|z|$. Additionally, this algorithm is vectorized, and an error testing framework is implemented. Finally, we implement the Faddeeva function by loading the toms_680 FORTRAN package into python.

2.1.2 Task: Write a vectorized function using the Taylor series algorithm.

The vectorized Taylor series function can be found in the attached erftools.py library. This library is loaded to the end of notebook for reference (see Appendix). Throughout this notebook, this function is called as mylib.erf_taylor.

2.1.3 Task: Compare results to special.erf and determine relative precision of the Taylor series algorithm for $|z| < 9$.

The first thing to determine is how many terms in the Taylor series must be used to obtain a desired precision throughout some region in the complex plane. To do this we will run through the coeffecients on a sparse array within the region $|z| < 9$ of the complex plane, and compute the maximum number of coefficients needed to ensure a precision of 10^{-10} throughout the entire region. This is the same precision that is garunteed in the scipy documentation references.

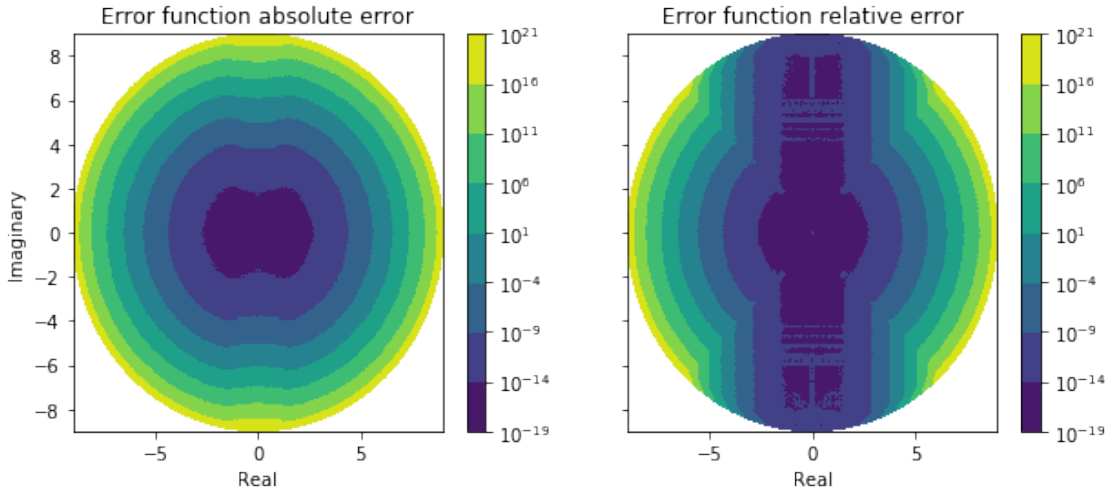
In [2]:
```
#Generate sparse grid of complex numbers contained in the disk of radius 9
x0,y0 = np.linspace(-9, +9, 201),np.linspace(-9, +9, 199)
xx0, yy0 = np.meshgrid(x0,y0)
test_grid = xx0 + 1.0j*yy0
max_n = 0

#Test number of coefficients needed for a 1e-10 precision throughout grid
for z in test_grid.flatten():

    term, n = z, 0

    #Iterate through Taylor coefficients until desired precision is found
    while term > 1e-10:

        n += 1
        term = (-1)**n * z**(2*n+1)/(special.factorial(n)*(2*n+1))

    #Track max number of coefficients needed
    if n > max_n:

        max_n = n

print('Maximum number of terms for a tolerance of 1e-10 is n='+str(max_n))
```

```
Maximum number of terms for a tolerance of 1e-10 is n=157
```

Next, we will look at the discrepency between the Taylor series algorithm and the function available in the scipy.special library in the given region. To ensure sufficient precision of the Taylor series, we will compute the entire grid up to the maximum of $n = 157$ terms.

```
In [3]: #Generate grid of complex numbers contained in the disk of radius 9
        x,y = np.linspace(-9, +9, 1000),np.linspace(-9, +9, 999)
        xx, yy = np.meshgrid(x,y)
        gridxy = xx + 1.0j*yy
        mask = (xx*xx+yy*yy < 9*9)*1

        #Apply error function to entire array
        z1 = special.erf(gridxy)
        z2 = mylib.erf_taylor(gridxy,nterms=max_n)

        #Compute absolute and relative error
        abs_diff = np.abs(z1-z2)
        rel_diff = abs_diff / np.abs(z1)
```

```
In [16]: #Generate contour plots of the discrepancy
         fig, axes = plt.subplots(1, 2, figsize=(10,4), sharey=True)

         plt.sca( axes[0] )
         plt.contourf(xx,yy,abs_diff*mask,norm=colors.LogNorm())
         plt.colorbar()
         plt.xlabel('Real')
         plt.ylabel('Imaginary')
         plt.title('Error function absolute error')

         plt.sca( axes[1] )
         plt.contourf(xx,yy,rel_diff*mask,norm=colors.LogNorm())
         plt.colorbar()
         plt.xlabel('Real')
         plt.title('Error function relative error')

         plt.show()
```



Note the that scale in each contour plot is logarithmic. We see that the Taylor series approach begins to diverge from the scipy.special.erf function exponentially as $|z|$ increases. Interestingly, the error relative to the value of the error function remains well behaved for values which are close to the imaginary axis. This suggests that the Taylor series algorithm becomes unreliable only when the real part of z gets large. In particular, the algorithm seems to be reliable to 9 significant digits whenever $\mathrm{Re}(z) < 4$.

Another comparison to make is with the speed of the computation.

```
In [5]: rand_array = np.random.random(999) + 1.0j*np.random.random(999)
        %timeit special.erf(rand_array)
        %timeit mylib.erf_taylor(rand_array,nterms = max_n)
```

```
245 ţs ś 9.75 ţs per loop (mean ś std. dev. of 7 runs, 1000 loops each)
6.24 ms ś 310 ţs per loop (mean ś std. dev. of 7 runs, 100 loops each)
```

In this case, all of the values computed lie within the unit disk of the complex plane. In this region, the values returned by both functions agree to double precision error, according to the previous plots. Although the values agree in this region, we see that the scipy.special.erf function is at least an order of magnitude quicker than the direct Taylor series approach.

2.1.4 Task: Write python code for some more complex algorithm that you find on the web.

The algorithm implemented here is based on the widely used Gautschi algorithm, which is used to compute the value of the Faddeeva function

$$w(z) = e^{-z^2}\left(1 - \mathrm{erf}(-iz)\right)$$

in the first quadrant of the complex plane. The value of the Faddeeva function in the remaning quadrants can be readily obtained using the sign inversion properties:

$$w(-z) = 2e^{-z^2} - w(z)$$

$$w(z^*) = [w(-z)]^*$$

By writing a function to compute $w(z)$, the error function can then be computed by inverting the above definition. By mapping $z \mapsto iz$, we obtain

$$erf(z) = 1 - e^{-z^2} w(iz)$$

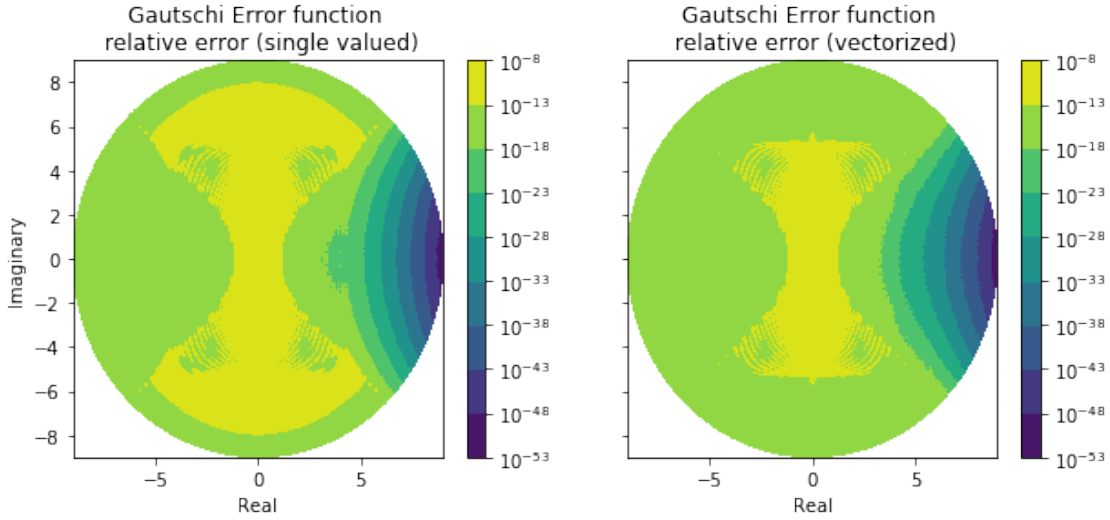
A full derivation of the Gautschi algorithm can be found in: *Efficient Computation of the Complex Error Function Walter Gautschi SIAM Journal on Numerical Analysis Vol. 7, No. 1 (Mar., 1970), pp. 187-198*, and the algorithm is described precisely in: https://dl.acm.org/citation.cfm?id=363618 .

I wrote two seperate implementations of the Gautschi algorithm (see Appendix). The first follows the cited paper precisely, but is limited to computing single values of the error function. Such a function can be vectorized in a "brute-force" type manner by simply looping over some input array. In an attempt to optimize this algorithm for a vectorized input, I made a second function where if/else statements are swapped with boolean arrays (to be used as logical masks), in an attempt to exploit the efficiency of NumPy array algebra. The downside to this is that part of the algorithm is recursive, and the number of iterations in this step is dependent on the input. Since the "boolean mask" approach requires acting on the entire input array at once, we must perform the recursive step on the entire array as many times as needed to ensure that every number in the array has been adequetly processed. For example, if an array of 5 numbers has for numbers that must be iterated twice through the recursive step, and one that must be iterated 10 times, the entire array must be iterated 10 times, leading to 8 redundant operations on the other four.

We start by comparing each of these functions to the one in the SciPy library.

```
In [6]: #Explicitely vectorize by looping over array
        erf_npvec = np.vectorize(mylib.erf)

        #Compute error function across grid
        z3 = erf_npvec(gridxy)
        z4 = mylib.erf_vec(gridxy)

        #Compare relative difference of each method to the special.erf function
        rel_diff3 = np.abs(z1-z3) / np.abs(z1)
        rel_diff4 = np.abs(z1-z4) / np.abs(z1)

In [7]: #Generate contour plots of the discrepancy
        fig, axes = plt.subplots(1, 2, figsize=(10,4), sharey=True)

        plt.sca( axes[0] )
        plt.contourf(xx,yy,rel_diff3*mask,norm=colors.LogNorm())
        plt.colorbar()
        plt.xlabel('Real')
        plt.ylabel('Imaginary')
        plt.title('Gautschi Error function \n relative error (single valued)')

        plt.sca( axes[1] )
        plt.contourf(xx,yy,rel_diff4*mask,norm=colors.LogNorm())
        plt.colorbar()
```

```
plt.xlabel('Real')
plt.title('Gautschi Error function \n relative error (vectorized)')

plt.show()
```



We see that each of these functions are comparable in relative difference to the one available in the SciPy library. Interestingly, it seems that the discrepency is the greatest near the origin, in complete contrast to the Taylor series algorithm. It is clear the the Gautschi algorithm is significantly more numerically stable, as is agrees with the SciPy one to at least 8 significant figures for all $|z| < 9$.

Given the tradeoff between exploiting the NumPy array algebra, and the possibility of having to perform redundant operations on some numbers, It will be interesting to see how the vectorized functions compare in computational time. Additionally, as the three of these functions agree for all values in the region of interest, the comparison between them will naturally come down to speed.

```
In [8]: rand_array = np.random.random(999) + 1.0j*np.random.random(999)
        %timeit special.erf(rand_array)
        %timeit erf_npvec(rand_array)
        %timeit mylib.erf_vec(rand_array)
```

```
240 ţs ś 2.87 ţs per loop (mean ś std. dev. of 7 runs, 1000 loops each)
93.4 ms ś 2.32 ms per loop (mean ś std. dev. of 7 runs, 10 loops each)
20.1 ms ś 819 ţs per loop (mean ś std. dev. of 7 runs, 10 loops each)
```

Despite my efforts, the SciPy special.erf function prevails, being almost two orders of magnitude quicker than my own variant of the Gautschi algorithm. More interesting to me is the fact that my approach of using mask arrays and NumPy array algebra to vectorize the Gautschi algorithm has led to roughly a factor of 5 speedup, in comparison to the brute force approach of looping over the input grid.

Task: Develop a testing framework for your error function A several unittest functions are included in the erftools.py library (see Appendix). The error testing framework ensures that each of the Gautschi functions generate the first 19 zeroes of the error function reliably. Additionally, the Taylor series algorithm is compared to the SciPy algorithm for 19 random values in the unit disk.

```
In [9]: %run erftools.py
```

```
test_ErfTaylor (__main__.TestErf) ... ok
test_ErfVecZeroes (__main__.TestErf) ... ok
test_ErfZeroes (__main__.TestErf) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.018s

OK


<Figure size 432x288 with 0 Axes>
```

Task: Use f2py to get a python interface to the TOMS fortran code The final function that we will look at is the Faddeeva function in the TOMS680 FORTRAN library, which can be loaded to python using f2py.

```
In [10]: %cd /home/alexander.hickey/PHYS581/Assignment 1/

         #Run fortran source code to produce wrapper
         import subprocess
         cmnd = r'/home/alexander.hickey/anaconda3/bin/f2py'
         cmnd += ' -c toms680_f2py.f'
         cmnd += ' -m toms680_f2py'
         subprocess.run( cmnd, shell=True, check=True)
```

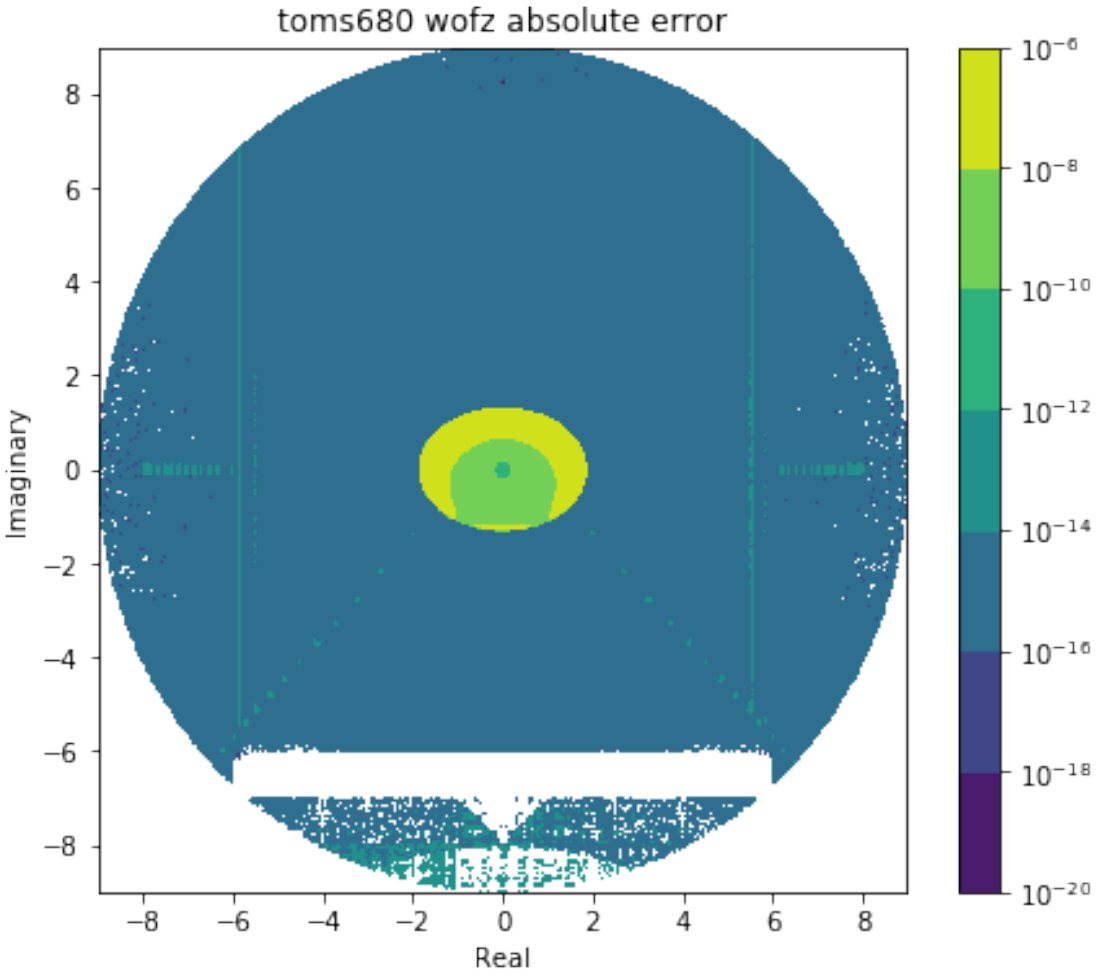
```
/home/alexander.hickey/PHYS581/Assignment 1


Out[10]: CompletedProcess(args='/home/alexander.hickey/anaconda3/bin/f2py -c toms680_f2py.f -m
```

As before, we vectorize the function and compare to the one in the SciPy library.

```
In [11]: #Load module
         import toms680_f2py as toms680

         #Vectorize the wofz function
         wofz_toms680 = np.vectorize(toms680.wofz)

         #Apply to grid and compare with SciPy function
         zw_scipy = special.wofz(gridxy)
```

```
        rezf, imzf, flag = wofz_toms680(np.real(gridxy),np.imag(gridxy))

        #Compute relative difference
        rel_diff_ft = np.abs(rezf+1.0j*imzf - zw_scipy)/np.abs(zw_scipy)
```

In [17]:
```
        #Generate contour plots of the discrepancy
        plt.figure(figsize = (7,6))
        plt.contourf(xx,yy,rel_diff_ft*mask,norm=colors.LogNorm())
        plt.colorbar()
        plt.xlabel('Real')
        plt.ylabel('Imaginary')
        plt.title('toms680 wofz absolute error')

        plt.show()
```



The whitespace in the contour plot corresponds to zeros on the input grid. It seems that the two functions agree in the region of interest. Just as with the Gautschi algorithm functions, the

discrepency is the greatest near the origin, but in this case the difference seems to be relatively uniform for larger $|z|$. Finally, we compare the time.

```
In [13]: rand_array = np.random.random(999) + 1.0j*np.random.random(999)
         %timeit special.wofz(rand_array)
         %timeit wofz_toms680(np.real(rand_array),np.imag(rand_array))

184 ţs ś 4.94 ţs per loop (mean ś std. dev. of 7 runs, 10000 loops each)
514 ţs ś 11.8 ţs per loop (mean ś std. dev. of 7 runs, 1000 loops each)
```

Once again, the scipy function prevails in terms of computational speed over a size 999 array. Compared to the Taylor series and Gautschi algorithm functions however, the FORTRAN wrapper has provided a significant speedup.

2.1.5 Conclusion

In this notebook, I investigated various approaches of computing the vectorized complex error function, in the region $|z| < 9$. I found that direct computation of the Taylor series tends to be numerically unstable, and thus unreliable, whenever $|z| > 4$. I implemented the Gautschi algorithm, and vectorized it both by looping over the input array (np.vectorize), and by modifying logical statements to exploit the efficiency of NumPy array algebra. Additionally, I used f2py, a FORTRAN wrapper to implement the Faddeeva function from the TOMS library. I found that each of these implementations agreed with the SciPy library to at least 8 significant figures in the region of interest. I also found that exploiting the array algebra led to roughly a factor of 5 speedup for computing the error function on large arrays in the unit disk, in comparison to looping over the array. Additionally, an error testing framework was implemented to ensure that the Taylor series function and Gautschi functions agree with the SciPy library functions. Finally, I found that the Faddeeva function available in the TOMS library is significantly quicker than the Taylor series and Gautschi algorithm approached, but still about 3 times slower than the one available in the SciPy library. Given more time, it would be interesting to see if one can obtain a significant speedup by explicitely vectorizing the TOMS function in FORTRAN, wrather than using the np.vectorize method which simply loops over the entire array.

Appendix: erftools.py

```
"""
Author: Alexander Hickey

This python library contains functions used to compute the complex
error function. A testing framework is built in, and can be run
by loading this file into the __main__ namespace.
"""

import numpy as np

def erf_taylor(z, nterms=19):
    '''
    Compute the standard error function using a Laurent series about z=0.

    Args:
        z: Array-like, collection of complex numbers
        nterms: Number of terms in the Taylor series to use

    Return:
        erf_z: Error function evaluated at each number
    '''

    #Cast to numpy array
    z = np.array(z)

    #Array to keep track of terms in series
    terms = np.zeros((nterms,)+z.shape, dtype=np.complex128)

    #Generate terms in series
    terms[0] = z
    z2 = -z*z
    for n in range(1,nterms):
        terms[n] = terms[n-1] * z2 / n * (2*n-1)/(2*n+1)

    #Sum over array of terms
    return np.sum(terms, axis=0) * 2.0 / np.sqrt(np.pi)



def wQ1(z):
    '''
    Compute the Faddeeva function in the first quadrant of the complex plane.
    Algorithm is garunteed an accuracy of at least 10 significant figures,
    see: https://dl.acm.org/citation.cfm?id=363618

    Args:
        z: Single complex number in the first quadrant (Re >= 0 and Im >=0)
```

```
    Return:
        w(z): Faddeeva function at z
    '''

    #Separate real and imaginary parts
    x, y = np.real(z), np.imag(z)

    #The remainder of the algorithm is described in:
    # https://dl.acm.org/citation.cfm?id=363618
    if y<4.29 and x<5.33:

        s=(1-y/4.29)*np.sqrt(1-x*x/28.41)
        h=1.6*s
        h2 = 2*h
        capn = int(6+23*s)
        nu = int(9+21*s)

    else:
        h, capn, nu = 0,0,8

    if h>0:
        lamb = h2**capn

    b = h==0 or lamb==0

    #Initialize parameters
    r1, r2, s1, s2 = 0,0,0,0

    for n in range(nu,-1,-1):

        np1 = n+1
        t1 = y+h+np1*r1
        t2 = x-np1*r2
        c= .5/(t1*t1+t2*t2)
        r1 = c*t1
        r2 = c*t2

        if h>0 and n<=capn:

            t1 = lamb + s1
            s1 = r1*t1-r2*s2
            s2 = r2*t1+r1*s2
            lamb = lamb/h2

    if y==0:
        re = np.exp(-x*x)
    else:
        re = 1.12837916709551*(r1*b+s1*(not b))

    im = 1.12837916709551*(r2*b+s2*(not b))

    return re+im*1.0j
```

```
101
102
103
104
105 def wQ1_vec(z):
106     '''
107     Compute the Faddeeva function in the first quadrant of the complex plane.
108     Algorithm is garunteed an accuracy of at least 10 significant figures,
109     see: https://dl.acm.org/citation.cfm?id=363618. Modified to handle vector
110     input.
111
112     Args:
113         z: Array of complex numbers in the first quadrant (Re >= 0 and Im >=0)
114
115     Return:
116         w(z): Faddeeva function at z
117     '''
118
119     #Separate real and imaginary parts
120     x, y = np.real(z), np.imag(z)
121
122     #The remainder of the algorithm is based on the algorithm:
123     #https://dl.acm.org/citation.cfm?id=363618
124     #Several modifications are made to use numpy array algebra by
125     #replacing if/else statements with boolean arrays
126
127     #Mask inner and outer regions of algorithm
128     inner = np.logical_and(y<4.29,x<5.33)*1.
129     outer = np.logical_not(inner)*1.
130
131     #Define algorithm parameter arrays
132     s= (1-y/4.29)*np.abs(np.sqrt(1-x*x/28.41+0.0j))*inner
133     h= (1.6*s)*inner
134     h2 = 2*h
135     capn = ((6+23*s)*inner+0*outer).astype(int)
136     nu = ((9+21*s)*inner +8*outer).astype(int)
137     lamb = (h2**capn)*((h>0)*1)
138     b = np.logical_or(h==0,lamb==0)
139
140     #Initialize a bunch of stuff
141     r1 = np.zeros(z.shape)
142     r2, s1, s2 = r1.copy(), r1.copy(), r1.copy()
143     t1, t2, c = r1.copy(), r1.copy(), r1.copy()
144
145     #Maximum index required to process entire array
146     nustart = np.max(nu)
147
148     for n in range(nustart,-1,-1):
149
150         #Update counter
151         np1 = n+1
152
153         #Only act on elements with positive counters
```

```
        mask_act = (nu>0)*1

        #Negate mask
        mask_ignore = np.logical_not(mask_act)*1

        #Update parameters
        t1 = (y+h+np1*r1)*mask_act + t1*mask_ignore
        t2 = (x-np1*r2)*mask_act + t2*mask_ignore
        c= .5/(t1*t1+t2*t2)*mask_act + c*mask_ignore
        r1 = (c*t1)*mask_act + r1*mask_ignore
        r2 = (c*t2)*mask_act+ r2*mask_ignore

        #More masks
        mask2 = np.logical_and(n<=capn,h>0)*1
        mask2not = np.logical_not(mask2)*1

        #Update parameters
        t1 = (lamb + s1)*mask2 + t1*mask2not
        s1 = (r1*t1-r2*s2)*mask2+ s1*mask2not
        s2 = (r2*t1+r1*s2)*mask2+ s1*mask2not
        lamb = (lamb/(h2+mask2not))*mask2+ lamb*mask2not

    #Result
    re =np.exp(-x*x)*(y==0)+1.12837916709551*(r1*b+s1*np.logical_not(b))*(y!=0)
    im = 1.12837916709551*(r2*b+s2*np.logical_not(b))

    return re+im*1.0j



def w(z):
    '''
    Compute the Faddeeva function in any quadrant of the complex plane, by
    mapping it to the first quadrant with symmetry properties.

    Args:
        z: Single complex number

    Return:
        w(z): Faddeeva function at z
    '''

    #Real and imaginary parts
    re, im = np.real(z), np.imag(z)

    #z in first quadrant
    if re >= 0 and im >= 0:

        return wQ1(z)

    #z in second quadrant
    elif re < 0 and im > 0:

```

```
        return np.conj(wQ1(-np.conj(z)))

    #z in third quadrant
    elif re <=0 and im <= 0:

        return 2*np.exp(-z*z)-wQ1(-z)

    #z in fourth quadrant
    else:

        return 2*np.exp(-z*z)-np.conj(wQ1(np.conj(z)))




def w_vec(z):
    '''
    Compute the Faddeeva function in any quadrant of the complex plane, by
    mapping it to the first quadrant with symmetry properties.
    Modified to handle vector input.

    Args:
        z: Array of complex numbers

    Return:
        w(z): Faddeeva function at z
    '''

    #Real and imaginary parts
    re, im = np.real(z), np.imag(z)

    #Masks corresponding to each quadrant
    m1 = np.logical_and(re > 0,im >= 0)
    m2 = np.logical_and(re <= 0,im > 0)
    m3 = np.logical_and(re < 0,im <= 0)
    m4 = np.logical_and(re >= 0,im < 0)
    m0 = np.logical_and(re == 0,im == 0) #Don't forget origin!

    #Compute the result for each quadrant using symmetry properties
    resQ1 = wQ1_vec(z)
    resQ2 = np.conj(wQ1_vec(-np.conj(z)))
    resQ3 = 2*np.exp(-z*z)-wQ1_vec(-z)
    resQ4 = 2*np.exp(-z*z)-np.conj(wQ1_vec(np.conj(z)))

    return resQ1*m1+resQ2*m2+resQ3*m3+resQ4*m4+1.0*m0


def erf(z):
    '''
    Compute the error function in the complex plane.
    Algorithm is garunteed an accuracy of at least 10 significant figures,
    see: https://dl.acm.org/citation.cfm?id=363618

```

```
    Args:
        z: Single complex number

    Return:
        erf(z): Error function at z
    '''

    return 1-np.exp(-z*z)*w(1.0j*z)


def erf_vec(z):
    '''
    Compute the error function in complex plane.
    Algorithm is garunteed an accuracy of at least 10 significant figures,
    see: https://dl.acm.org/citation.cfm?id=363618. Modified to handle vector
    input.

    Args:
        z: Array of complex numbers

    Return:
        erf(z): Error function at z
    '''

    return 1.0-np.exp(-z*z)*w_vec(1.0j*z)




################################################################################
#Testing framework

import unittest
import scipy.special as special

class TestErf(unittest.TestCase):
    '''
    Unit testing class for functions in the erftools.py library
    '''

    def test_ErfTaylor(self):
        '''
        Test that the Maclaurin series converges to the error function
        within the unit disk. Tolerance is set to 1e-10 for n=19 terms.
        '''
        #Generate random array of 19 numbers in the unit disk.
        xr, yr = np.random.random(19), np.random.random(19)
        zr = xr + 1.0j*yr

        #Compare to special.erf
        diff = np.abs(erf_taylor(zr)-special.erf(zr))

        self.assertTrue( np.any(diff<1e-10) )
```

```
313
314     def test_ErfZeroes(self):
315         '''
316         Test that the first 19 zeroes of the erf function correspond
317         with the zeroes generated by the scipy.special.erf_zeros
318         function.
319         '''
320         #Compute first 19 zeroes
321         z0 = special.erf_zeros(19)
322         erf0 = np.abs([erf(z0[j]) for j in range(len(z0))])
323 
324         #Assert that |erf(z0)-0| < 1e-10 for all z0
325         #This is the tolerance garunteed by the
326         #Gautschi algorithm.
327         self.assertTrue( np.any(erf0<1e-10) )
328 
329     def test_ErfVecZeroes(self):
330         '''
331         Test that the first 19 zeroes of the erf_vec function correspond
332         with the zeroes generated by the scipy.special.erf_zeros
333         function.
334         '''
335         #Compute first 19 zeroes
336         z0 = special.erf_zeros(19)
337         erf0 = np.abs(erf_vec(z0))
338 
339         #Assert that |erf(z0)-0| < 1e-10 for all z0
340         #This is the tolerance garunteed by the
341         #Gautschi algorithm.
342         self.assertTrue( np.any(erf0<1e-10) )
343 
344 
345 #Run tests if in main namespace
346 if __name__ == '__main__':
347     unittest.main(argv=[''],verbosity=2,exit=False)
```