# Homework 6

By: Keito Watanabe (s6kewata), Haveesh Singirikonda (s6gusing)

In our code, we utilize the wavefunctions that are obtained from Dr. Nogga's code regarding the two-body bound state problem. We perform the following (omit the primes in $p$ for brevity):

1. Interpolate the wavefunctions to those at $|\vec{p} - \frac{1}{2}\vec{q}|$
2. Obtain the directions and rewrite them in terms of angles using the z-component (since $z = r\cos\theta$)
3. Evaulate the spherical harmonics at both $\hat{p}$ and $\hat{\vec{p} - \frac{1}{2}\vec{q}}$
4. Perform Gauss-Legendre quadrature twice to evaluate the angular and radial integrals to get the form factor.

In [1]:
```
import numpy as np
import math as m
from numpy.polynomial.legendre import leggauss
from scipy.special import legendre

class OBEpot:
    """Provides a method for the partial wave representation of the C

       The matrix elements are obtained by numerical intergration.
       The mass of the exchanged boson, the strength of the
       interaction and the couter term is given on initialization.
       The interaction is regularized using a cutoff that is also
       given on init.
    """

    # this are common parameters for all instances
    hbarc=197.327

    # init interaction
    def __init__(self, cutoff=500.0, C0=1.0, nx=12,mpi=138.0,A=-1.0):
        """Defines the one boson exchange for a given regulator, coup

        Parameters:
        cutoff -- regulator in MeV
        C0 -- strength of the short distance counter term (in s-wave)
        A -- strength of OBE
        nx -- number of angular grid points for numerical integratior
        mpi -- mass of exchange boson in MeV"""

        self.mpi = mpi/self.hbarc
        self.cutoff = cutoff/self.hbarc
        self.C0=C0
        self.A=A
        self.nx=nx

        self.xp=np.empty((self.nx),dtype=np.double)
        self.xw=np.empty((self.nx),dtype=np.double)
        self.xp,self.xw=leggauss(self.nx)
```

```
    # function defines the x integral
    def _g(self,pp,p,k):
        """Calculates g function of the partial wave decomposition of

           pp -- outgoing momentum
           p -- incoming momentum
           k -- angular momentum"""

        # define prefact
        # get the corresponding legendre polynomial
        Pk = legendre(k)
        # define momentum transfer dependent on angles
        qval=np.sqrt(p**2+pp**2-2*p*pp*self.xp)

        # build integral of regularized OBE
        return float(np.sum(Pk(self.xp)/((qval**2+self.mpi**2))*self.

    # determines complete, regularized interaction
    def v(self,pp,p,l):
        """Potential matrix element in fm**2

           pp -- outgoing momentum in fm**-1
           p -- incoming momentum in fm**-1
           l -- angular momentum"""

        # first overall prefact of 1pi exchange part  (cancel 2pi fac
        prefact=self.A

        mat=prefact*self._g(pp,p,l)

        if (l==0):   # add s-wave counter term
          mat+=self.C0*np.exp(-(pp**2+p**2)/self.cutoff**2)  # 4pi is

        return mat
```

In [2]:

```
import numpy as np
from numpy.polynomial.legendre import leggauss
from scipy.special import spherical_jn
from scipy.interpolate import interp1d
import timeit

class TwoBody:
    """Methods to obtain eigenvalues and eigenvectors for the bound s
    # define hbarc for unit conversion
    hbarc=197.327

    def __init__(self, pot, np1=20, np2=10, pa=1.0, pb=5.0, pc=20.0,
                            nr1=20, nr2=10, ra=1.0, rb=5.0, rc=20.0,
                            np1four=200,np2four=100):
        """Initialization of two-body solver.

           The initialization defines the momentum grids and the inte
           At this time, also the grid for Fourier transformation and
           wave function is given.

           Parameters:
```

```
        pot -- object that defines the potential matrix elements (
        np1 -- number of grid points in interval [0,pb]
        np2 -- number of grid points in interval [pb,pc]
        pa  -- half of np1 points are in interval [0,pa]
        pb  -- interval boundary as defined above
        pc  -- upper integration boundary for the solution of the
        mred -- reduces mass of the two bosons in MeV

        nr1 -- number of r points in interval [0,rb]
        nr2 -- number of r points in interval [rb,rc]
        ra  -- half of np1 points are in interval [0,pa]
        rb  -- interval boundary as defined above
        rc  -- upper integration boundary for the solution of the

        np1four -- number of p points in interval [0,pb] for Fouri
        np2four -- number of p points in interval [pb,pc] for Four

    # measure also time for preparation (mostly from potential)
    self.preptime=-timeit.default_timer()

    # store parameters (if necessary convert to fm)
    self.np1 = np1
    self.np2 = np2
    self.npoints  = np1+np2
    self.mred=mred/self.hbarc
    self.pa=pa
    self.pb=pb
    self.pc=pc
    self.l=l

    self.nr1 = nr1
    self.nr2 = nr2
    self.nrpoints  = nr1+nr2
    self.ra=ra
    self.rb=rb
    self.rc=rc

    self.np1four = np1four
    self.np2four = np2four
    self.npfour  = np1four+np2four

    # store grid points and weights for integral equations
    self.pgrid,self.pweight = self._trns(self.np1,self.np2,self.p

    # store grid points and weights for r space wave functions
    self.rgrid,self.rweight = self._trns(self.nr1,self.nr2,self.r

    # store grid points and weights for Fourier trafo
    self.pfourgrid,self.pfourweight = self._trns(self.np1four,sel

    # store underlying interaction
    self.pot=pot

    # and actual potential matrix elements
    self.vmat=np.empty((self.npoints,self.npoints),dtype=np.doubl
    for i in range(self.npoints):
      for j in range(self.npoints):
        self.vmat[i,j]=self.pot.v(self.pgrid[i],self.pgrid[j],sel

    self.preptime+=timeit.default_timer()
```

```
        print("Preparatione time: {0:15.6e} sec".format(self.preptime

        # fix timer for solution of the eigenvalue equations
        self.runtime=0.0

    def _trns(self,np1,np2,pa,pb,pc):
      """Auxilliary method that provides transformed Gauss-Legendre g

         This is using a hyperbolic trafo shown in the lecture.
         Parameter:
         np1 --  grid points in ]0,pb[
         np2 --  grid points are distributed in ]pb,pc[ using a linea

         pa  -- half of np1 points are in interval [0,pa]
         pb  -- interval boundary as defined above
         pc  -- upper integration boundary """

      x1grid,x1weight=leggauss(np1)
      x2grid,x2weight=leggauss(np2)

      # trafo (1.+X) / (1./P1-(1./P1-2./P2)*X) for first interval
      p1grid=(1.+x1grid) / (1./pa-(1./pa-2./pb)*x1grid)
      p1weight=(2.0/pa-2.0/pb)*x1weight / (1./pa-(1./pa-2./pb)*x1grid

      # linear trafo
      p2grid=(pc+pb)/2.0 + (pc-pb)/2.0*x2grid
      p2weight=(pc-pb)/2.0*x2weight

      pgrid=np.empty((self.npoints),dtype=np.double)
      pweight=np.empty((self.npoints),dtype=np.double)

      pgrid = np.concatenate((p1grid, p2grid), axis=None)
      pweight = np.concatenate((p1weight, p2weight), axis=None)

      return pgrid,pweight

# set up set of equations and calculate eigenvalues

    def eigv(self,E,neigv):
      """Solve two-body integral equation and return n-th eigenvalue,

         Parameters:
         E -- energy used in the integral equation in fm**-1
         neigv -- number of the eigenvalue to be used"""

    # measure timeing (compare for loop and einsum)
      self.runtime-=timeit.default_timer()

    # set up the matrix amat for which eigenvalues have to be calcula

      amat=np.einsum('i,ij,j->ij', 1.0/(E-self.pgrid**2/(2*self.mred)
      # replaces less performant for loops
      #amat=np.empty((self.npoints,self.npoints),dtype=np.double)
      #for i in range(self.npoints):
      #  for j in range(self.npoints):
      #    amat[i,j]=np.real(1.0/(E-self.pgrid[i]**2/(2*self.mred))*s

    # determine eigenvalues using numpy's eig method
      evalue,evec=np.linalg.eig(amat)
```

```
    # I now assume that the relevant eigenvalues are real to avoid co
      evalue=np.real(evalue)

    # remove neigv-1 largest eigenvalues
      for n in range(neigv-1):
        maxpos=np.argmax(evalue)
        evalue[maxpos]=0.0

    # take the next one
      maxpos=np.argmax(evalue)
      eigv=evalue[maxpos]
    # define solution as unnormalized wave function
      wf=evec[:,maxpos]
    # and normalize
      norm=np.sum(wf**2*self.pweight[0:self.npoints]*self.pgrid[0:sel
      wf=1/np.sqrt(norm)*wf

    # measure timeing (compare for loop and einsum)
      self.runtime+=timeit.default_timer()

      return eigv,self.pgrid[0:self.npoints],wf


    def esearch(self,neigv=1,e1=-0.01,e2=-0.0105,elow=0.0,tol=1e-8):
        """Perform search for energy using the secant method.

           Parameters:
           neigv -- number of the eigenvalue to be used
           e1 -- first estimate of binding energy (should be negative
           e2 -- second estimate of binding energy (should be negativ
           elow -- largest energy to be used in search (should be neg
           tol -- if two consecutive energies differ by less then tol

           Energies are given in fm**-1. """

        # determine eigenvalues for starting energies
        eta1,pgrid,wf=self.eigv(e1,neigv)
        eta2,pgrid,wf=self.eigv(e2,neigv)

        while abs(e1-e2) > tol:
          # get new estimate (taking upper value into account)
          enew=e2+(e1-e2)/(eta1-eta2)*(1-eta2)
          enew=min(elow,enew)

          # get new eigenvalue and replace e1 and e2 for next iterati
          eta,pgrid,wf=self.eigv(enew,neigv)
          e2=e1
          eta2=eta1
          e1=enew
          eta1=eta

        print("Total runtime: {0:15.6e} sec".format(self.runtime))

        return e1,eta1,pgrid,wf

    def fourier(self,wfp):
        """Calculates the Fourier transform of the partial wave repre

           Parameter:
```

```
            wfp -- wave function in momentum space

            Note that the factor I**l is omitted."""

        # calculate spherical bessel functions based dense Fourier tr
        # prepare matrix based on r,p points
        rpmat = np.outer(self.rgrid,self.pfourgrid)
        # evaluate jl
        jlmat = spherical_jn(self.l,rpmat)

        # interpolate of wave to denser Fourier trafo grid
        wfinter = interp1d(self.pgrid, wfp, kind='cubic',fill_value='
        # interpolate wf and multiply my p**2*w elementwise
        wfdense = wfinter(self.pfourgrid)*self.pfourgrid**2*self.pfou

        # now the Fourier trafo is a matrix-vector multiplication
        wfr = jlmat.dot(wfdense)

        return self.rgrid,wfr


    def rms(self,wfr):
        """Calculates the norm and rms radius for the given r-space w

            Normalization of the wave function is assumed.
            Parameter:
            wfr -- wave function in r-space obtained by previous Fouri


        norm=np.sum(wfr**2*self.rweight*self.rgrid**2)
        rms=np.sum(wfr**2*self.rweight*self.rgrid**4)


        rms=np.sqrt(rms)

        return norm,rms
```

In [3]:

```
import numpy as np
# prepare interpolation using cubic hermitian splines

class Cubherm:
    """Prepares spline functions for cubic hermitian splines.

    see Hueber et al. FBS 22,107 (1997).

    The function spl returns the the spline function for a given x.
    If x is below the smallest grid point, extrapolation is used.
    If x is after largest grid point, then the function evaluates to
    """


    def spl(xold,xin):
        """Calculates spline functions for given values xold and xnew

            Parameters:
            xold -- set of grid points where function is given. xold r
            xnew -- set of grid points to interpolate to. xnew can be
```

```
    On return spline functions will be given that have the sha
"""

# first determine the base value of the index for each xnew.

nold=len(xold)
if nold<4:
  raise(ValueError("Interpolation requires at least 4 grid po

xnew=xin.reshape((-1))
indx=np.empty((len(xnew)),dtype=int)

for i in range(len(xnew)):
  # do not extrapolated beyond largest grid point
  if xnew[i] > xold[nold-1]:
    indx[i]=-1
  else:
    for j in range(nold):
      if xnew[i] <= xold[j]:
        break
    if j < 1:
      indx[i]=0
    elif j > nold-3:
      indx[i]=nold-3
    else:
      indx[i]=j-1

# then prepare phi polynomials for each x

phi1=np.zeros((len(xnew)),dtype=np.double)
phi2=np.zeros((len(xnew)),dtype=np.double)
phi3=np.zeros((len(xnew)),dtype=np.double)
phi4=np.zeros((len(xnew)),dtype=np.double)

for i in range(len(xnew)):
  if indx[i]>0:
    phi1[i] = (xold[indx[i] + 1] - xnew[i]) ** 2 / (xold[indx
    phi2[i] = (xold[indx[i]] - xnew[i]) ** 2 / (xold[indx[i]
    phi3[i] = (xnew[i] - xold[indx[i]]) * (xold[indx[i] + 1]
    phi4[i] = (xnew[i] - xold[indx[i] + 1]) * (xold[indx[i]]

# now we are ready to prepare the spline functions
# most are zero
splfu=np.zeros((len(xold),len(xnew)),dtype=np.double)
for i in range(len(xnew)):
  if indx[i]>0:
    splfu[indx[i]-1,i] = \
       -phi3[i]*(xold[indx[i]+1]-xold[indx[i]])/(
                (xold[indx[i]]-xold[indx[i]-1])*(xold[indx[i]

    splfu[indx[i],i] = phi1[i] \
        +phi3[i]*((xold[indx[i]+1]-xold[indx[i]])/ (xold[indx
                 -(xold[indx[i]]-xold[indx[i]-1])/ (xold[indx
        -phi4[i]*(xold[indx[i]+2]-xold[indx[i]+1])/ (xold[ind

    splfu[indx[i]+1,i] = phi2[i] \
        +phi3[i]*(xold[indx[i]]-xold[indx[i]-1])/ (xold[indx[
        +phi4[i]*((xold[indx[i]+2]-xold[indx[i]+1])/ (xold[in
                 -(xold[indx[i]+1]-xold[indx[i]])/ (xold[indx
```

```
            splfu[indx[i]+2,i] = \
                phi4[i]*(xold[indx[i]+1]-xold[indx[i]])/(
                        (xold[indx[i]+2]-xold[indx[i]+1])*(xold[indx[
          elif indx[i]>=0:
            # do linear interpolation at the origin
            splfu[indx[i],i] = (xnew[i]-xold[indx[i]+1])/(xold[indx[i
            splfu[indx[i]+1,i] = (xold[indx[i]]-xnew[i])/(xold[indx[i

        retshape=[nold]
        for n in list(np.shape(xin)):
          retshape.append(n)

        return splfu.reshape(retshape)
```

In [4]:
```
# libaries for plotting
# numpy is used for the solution anyhow
import matplotlib.pyplot as plt
import numpy as np

# # get figure enviroment
# fig, ax = plt.subplots()

# # set labels
# ax.set_xlabel(r'$p$[fm$^{-1}$]')
# ax.set_ylabel(r'$\psi(p)$[fm$^{3/2}$]')

# # set some limits on axes to concentrate on most relevant part of t

# ax.set_xlim(0,8)
# ax.set_ylim(1E-6,15)


# fit result copied from above
res=[[300.0, -0.09827953494014054],
 [400.0, -0.028203145146196713],
 [500.0, -0.0004221894040945335],
 [600.0, 0.012857431330421717],
 [700.0, 0.020167185806378923],
 [800.0, 0.024707945457255083],
 [900.0, 0.027865200396659445],
 [1000.0, 0.030308007813785776],
 [1100.0, 0.03239034331482156],
 [1200.0, 0.034316113574472931]]

# we only want to extract the wavefunctions and the momentum grid fro
output_arr = []

# go through parameters of fit
for para in res:
    # determine interacion and set up solver
    pot=OBEpot(nx=24,mpi=138.0,C0=para[1],A=-1.0/6.474860194946856,cu
    solver=TwoBody(pot=pot,np1=40,np2=20,pa=1.0,pb=7.0,pc=35.0,mred=9
                           nr1=40, nr2=20, ra=1.0, rb=5.0, rc=20.0,
                           np1four=400,np2four=200)

    # perform energy search for this parameter set
    ener,lam,pmom,wf=solver.esearch(neigv=1,e1=-2.0/TwoBody.hbarc,e2=

    # get Fourier trafo (together with r grid points)
```

```
    rp,wfr=solver.fourier(wf)
    # calculate norm and rms radius (deviation of the norm from 1 giv
    norm,rms=solver.rms(wfr)

    # append the wavefunction, momentum grid, and rms
    output_arr.append([wf, pmom, rms])

#     # plot the wave function in momentum space (here use log scale
#     # of the sign of wf when a linear scale and no abs values is us
#     if wf[0]>0:
#       ax.semilogy(pmom,np.abs(wf),label=r"$\Lambda=$ {0:10.3f}".for
#     else:
#       ax.semilogy(pmom,np.abs(-wf),label=r"$\Lambda=$ {0:10.3f}".fo


#     # print energy, eigenvalue (should be one), norm and 1/2 rms (=
#     print("{0:15.6e}   {1:15.6e}  {2:15.6e}   {3:15.6e}".format(ene

# ax.legend(loc="best")

# plt.show()
```

```
Preparatione time:    6.983249e-01 sec
Total runtime:    6.740474e-03 sec
Preparatione time:    7.497770e-01 sec
Total runtime:    6.768524e-03 sec
Preparatione time:    6.823969e-01 sec
Total runtime:    7.200346e-03 sec
Preparatione time:    6.736867e-01 sec
Total runtime:    7.061575e-03 sec
Preparatione time:    6.759430e-01 sec
Total runtime:    6.416101e-03 sec
Preparatione time:    6.817197e-01 sec
Total runtime:    6.800667e-03 sec
Preparatione time:    6.659459e-01 sec
Total runtime:    6.920635e-03 sec
Preparatione time:    6.778718e-01 sec
Total runtime:    7.026191e-03 sec
Preparatione time:    6.807744e-01 sec
Total runtime:    7.052798e-03 sec
Preparatione time:    7.249987e-01 sec
Total runtime:    6.872474e-03 sec
```

In [5]:
```
def transf_leggauss(np1=20, np2=10, pa=1.0, pb=5.0, pc=20.0):
  """Auxilliary method that provides transformed Gauss-Legendre grid

     This is using a hyperbolic trafo shown in the lecture.
     Parameter:
     np1 --  grid points in ]0,pb[
     np2 --  grid points are distributed in ]pb,pc[ using a linear tr

     pa  -- half of np1 points are in interval [0,pa]
     pb  -- interval boundary as defined above
     pc  -- upper integration boundary """

  x1grid,x1weight=leggauss(np1)
  x2grid,x2weight=leggauss(np2)

  # trafo (1.+X) / (1./P1-(1./P1-2./P2)*X) for first interval
  p1grid=(1.+x1grid) / (1./pa-(1./pa-2./pb)*x1grid)
```

```
    p1weight=(2.0/pa-2.0/pb)*x1weight / (1./pa-(1./pa-2./pb)*x1grid)**2

    # linear trafo
    p2grid=(pc+pb)/2.0 + (pc-pb)/2.0*x2grid
    p2weight=(pc-pb)/2.0*x2weight

    pgrid=np.empty((np1+np2),dtype=np.double)
    pweight=np.empty((np1+np2),dtype=np.double)

    pgrid = np.concatenate((p1grid, p2grid), axis=None)
    pweight = np.concatenate((p1weight, p2weight), axis=None)

    return pgrid,pweight
```

In [6]:
```
# interpolate the wavefunction to the shifted momentum space

def interp_wf(wf, q, pgrid, nx_grid=40):
    '''Interpolate psi(p) -> psi(|p - 0.5 q|)'''

    # evaluate the new momentum grid
    x = np.linspace(-1, 1, nx_grid)

    P, X = np.meshgrid(pgrid, x)
    pgrid_new = np.sqrt(P**2. - 0.5*P*q*X + 0.25*q**2.)

    # evaluate the spline function
    spl_fcn = Cubherm.spl(pgrid, pgrid_new)

    # evaluate the new wavefunction
    wf_new = np.sum([spl_fcn[i,:,:] * wf[i] for i in range(len(wf))],

    return pgrid_new, wf_new

def eval_costhetas(q, pgrid_new, pgrid, nx_grid=40):
    '''
    Evaluate the direction of vec(p), vec(p - 0.5*q).
    Solely taken from the z component.
    '''
    # original costheta is x in [-1, 1]
    x = np.linspace(-1, 1, nx_grid)
    P, X = np.meshgrid(pgrid, x)

    # new theta taken from the new pgrid
    # z component is given byt px - 0.5*q
    costheta_new = (P*X - 0.5*q) / pgrid_new

#     print(costheta_new)

    # constrain costheta to be in [-1,1]
    for i, costheta_rows in enumerate(costheta_new):
        costheta_rows[costheta_rows > 1] = 2 - costheta_rows[costheta
        costheta_rows[costheta_rows < -1] = 2 + costheta_rows[costhet

    # return both old and new theta
    return X, costheta_new
```

In [7]:
```
# evaluate the integral
```

```
from scipy.special import legendre, lpmv
import math
from numpy.polynomial.legendre import leggauss

def Ylm(x, l, lz):
    '''Spherical harmonics (l, lz) without azimuthal component'''
    prefactor = np.sqrt((2*l + 1) / (4*np.pi) * math.factorial(l - lz
    # evaluate the associated legendre polynomial
    assos_leg = lpmv(lz, l, x)
    return prefactor * assos_leg

def eval_Fx(q, l, lz, wf, wf_new, pgrid_new, pgrid):
    '''Evaluate form factor in angular component'''

    # get direction of vec(p), vec(p - 0.5*q)
    costheta, costheta_new = eval_costhetas(q, pgrid_new, pgrid)

    # evaluate spherical harmonics at x = cos(theta)
    # difference in direction fully determined by z component
    # since variation in x component is not present
    ylm_p = Ylm(costheta, l, lz)  # old one
    ylm_pq = Ylm(costheta_new, l, lz)  # new one

    # evaluate the costheta integral
    # need wavefunctions since new one depends on costheta
    # by gauss-legendre quadrature
    nx_grid = pgrid_new.shape[1]  # only integrate over angular compo
    _, x_weights = leggauss(nx_grid)
    Fx = np.sum(np.conjugate(ylm_p) * ylm_pq * np.conjugate(wf) * wf_

    return Fx


def eval_Fq2(Fx, pa=1.0, pb=5.0, pc=20.0):
    '''Evaluate the form factor'''

    # seperate number of points to (approx). half and half
    N_fx = len(Fx)
    n_fx_1 = N_fx // 2
    n_fx_2 = N_fx - n_fx_1

    # get transformed grid
    pgrid, pweight = transf_leggauss(n_fx_1, n_fx_2, pa, pb, pc)

    # evaluate integral
    F = 2. * np.pi * np.sum(pweight * pgrid**2. * Fx)

    return np.real(F)
```

In [8]:
```
# test cell

q = 0
pgrid = output_arr[-1][1]
wf = output_arr[-1][0]
nx_grid=40

F_arr = np.zeros(10)

pgrid_new, wf_new = interp_wf(wf, q, pgrid)
# print(wf_new)
```

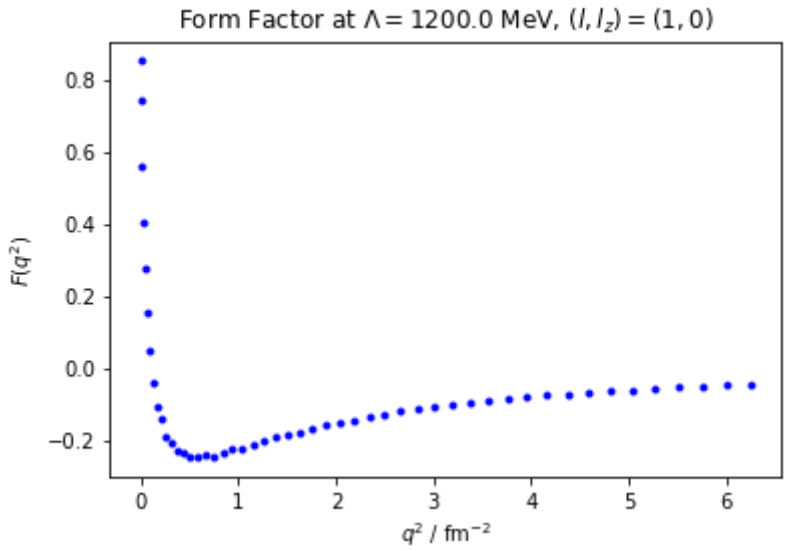
```
theta, costheta_new = eval_costhetas(q, pgrid_new, pgrid)
# Ylm(costheta_new, 1, 1)
# costheta_new

fx = eval_Fx(q, 1, 0, wf, wf_new, pgrid_new, pgrid)
# print(fx)
eval_Fq2(fx, pa=0.55, pb=1., pc=20.0)
# wf_new
```

Out[8]: 0.9926596482257762

In [9]:
```
Nq = 50
q_arr = np.linspace(0.001, 2.5, Nq)
wf, pgrid, _ = output_arr[-1]
nx_grid=40
l, lz = 1, 0   # specific angular momentum state

F_arr = np.zeros(Nq)

for i, q in enumerate(q_arr):
    pgrid_new, wf_new = interp_wf(wf, q, pgrid)

    theta, costheta_new = eval_costhetas(q, pgrid_new, pgrid)
    fx = eval_Fx(q, l, lz, wf, wf_new, pgrid_new, pgrid)
    # played around values until we get F(0) = 1
    F_arr[i] = eval_Fq2(fx, pa=0.55, pb=1., pc=20.0)
# wf_new
```

In [10]:
```
fig, ax = plt.subplots(figsize=(6,4))
ax.plot(q_arr**2., F_arr, marker="o", ls="", color="b", ms=3.0)
ax.set_xlabel("$q^2$ / fm$^{{-2}}$")
ax.set_ylabel("$F(q^2)$")
ax.set_title("Form Factor at $\Lambda = {0}$ MeV, $(l, l_z) = ({1},
```

Out[10]: Text(0.5, 1.0, 'Form Factor at $\\Lambda = 1200.0$ MeV, $(l, l_z) = (1, 0)$')



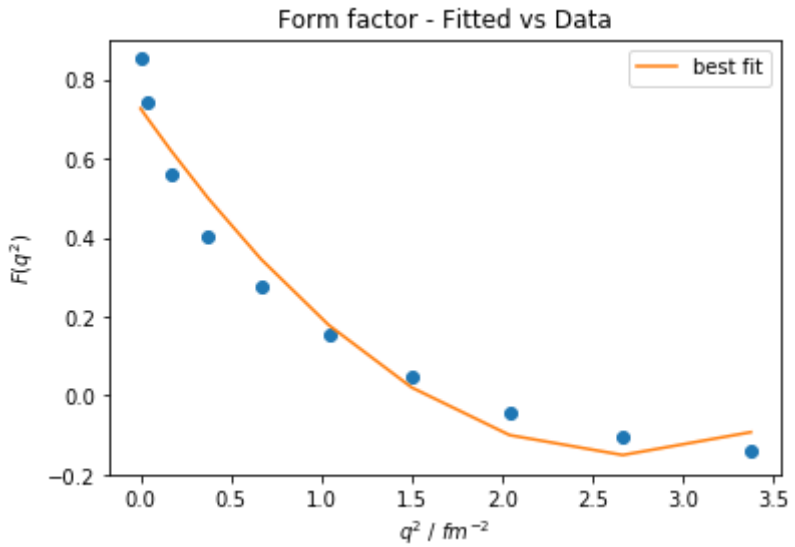

In [33]:
```
# fit F with quadratic polynomial by Taylor expanding F(q^2)
from lmfit import Model

def F_taylor(q2, a, b, c):
```

```
    '''Taylor expansion of F at low momentum transfer values'''
    return c + q2 * b + a / 2 * q2**2.

# test
rms_0 = np.real(output_arr[-1][2])

F_model = Model(F_taylor)
params = F_model.make_params(a = 0.2, b = -1/6*rms_0**2, c = 1)
q2_arr_trunc = q_arr[q_arr <= 2]**2.
F_arr_trunc = np.real(F_arr[q_arr <= 2])

result = F_model.fit(F_arr_trunc, params, q2=q2_arr_trunc)
```

In [34]:
```
print(result.fit_report())
```

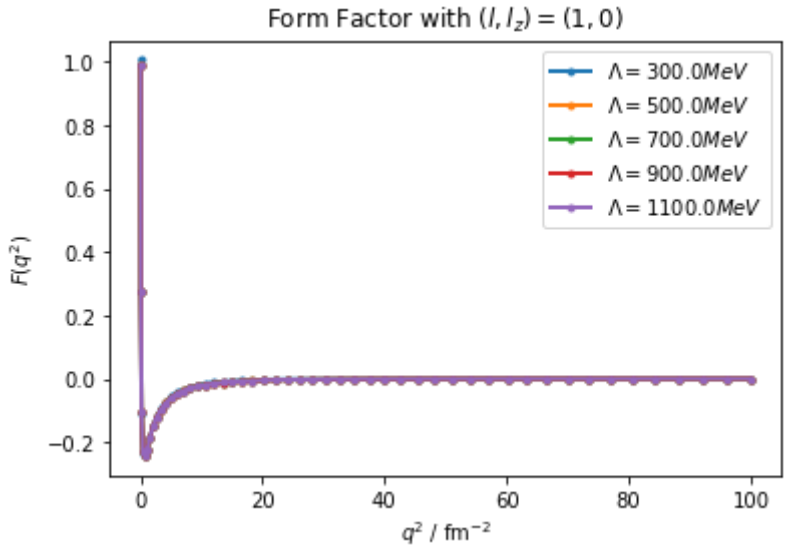
```
[[Model]]
    Model(F_taylor)
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 8
    # data points      = 10
    # variables        = 3
    chi-square         = 0.04457011
    reduced chi-square = 0.00636716
    Akaike info crit   = -48.1327687
    Bayesian info crit = -47.2250134
[[Variables]]
    a:  0.24326504 +/- 0.04870461 (20.02%) (init = 0.2)
    b: -0.65296411 +/- 0.07942929 (12.16%) (init = -2.858985)
    c:  0.72616280 +/- 0.04484147 (6.18%) (init = 1)
[[Correlations]] (unreported correlations are < 0.100)
    C(a, b) = -0.959
    C(b, c) = -0.718
    C(a, c) = 0.572
```

In [35]:
```
plt.plot(q2_arr_trunc, F_arr_trunc, 'o')
# plt.plot(q2_arr_trunc, result.init_fit, '--', label='initial fit')
plt.plot(q2_arr_trunc, result.best_fit, '-', label='best fit')
plt.legend()
plt.xlabel("$q^2$ / $fm^{{-2}}$")
plt.ylabel("$F(q^2)$")
plt.title("Form factor - Fitted vs Data")
plt.show()
```



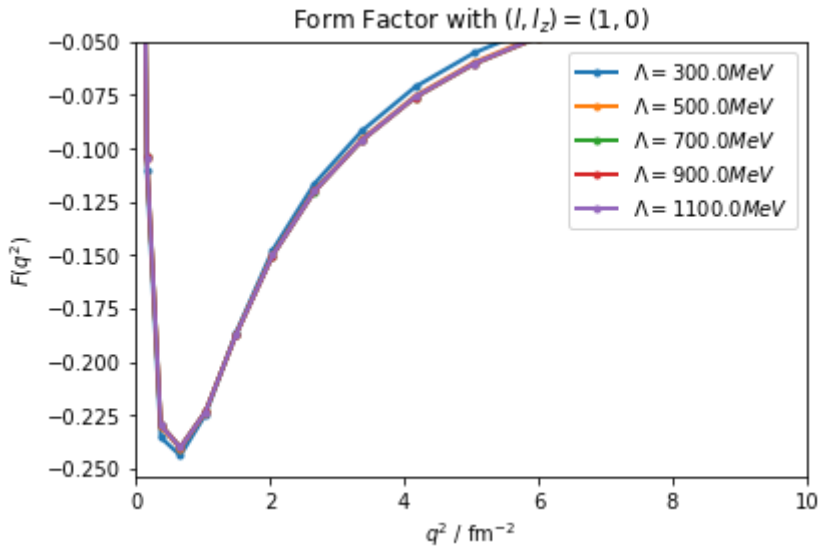
In [19]:
```
# now plotting for differnet values of Lambda
Nq = 50
q_arr = np.linspace(0, 10, Nq)
nx_grid=40
l, lz = 1,0   # specific angular momentum state

F_arr_lmbda = np.zeros((len(res), Nq))

for i, (Lmbda, _) in enumerate(res[::2]):

    wf, pgrid, rms = output_arr[i]

    for j, q in enumerate(q_arr):
        pgrid_new, wf_new = interp_wf(wf, q, pgrid)

        theta, costheta_new = eval_costhetas(q, pgrid_new, pgrid)
        fx = eval_Fx(q, l, lz, wf, wf_new, pgrid_new, pgrid)
        # played around values until we get F(0) = 1
        F_arr_lmbda[i,j] = eval_Fq2(fx, pa=0.55, pb=1., pc=20.0)
    # wf new
```

In [20]:
```
fig, ax = plt.subplots(figsize=(6,4))
for i, (Lmbda, _) in enumerate(res[::2]):
    ax.plot(q_arr**2., F_arr_lmbda[i,:], marker="o", ls="-", lw=2.0,
ax.set_xlabel("$q^2$ / fm$^{{-2}}$")
ax.set_ylabel("$F(q^2)$")
ax.set_title("Form Factor with $(l, l_z) = ({1}, {2})$".format(0, l,
ax.legend();
```



In [21]:
```
fig, ax = plt.subplots(figsize=(6,4))
for i, (Lmbda, _) in enumerate(res[::2]):
    ax.plot(q_arr**2., F_arr_lmbda[i,:], marker="o", ls="-", lw=2.0,
ax.set_xlabel("$q^2$ / fm$^{{-2}}$")
ax.set_ylabel("$F(q^2)$")
ax.set_title("Form Factor with $(l, l_z) = ({1}, {2})$".format(0, l,
ax.set_xlim([0,10])
ax.set_ylim([np.min(F_arr_lmbda)-0.01, -0.05])
ax.legend()
```

Out[21]:
```
<matplotlib.legend.Legend at 0x7f12dc342160>
```




We observe that with lower values of $\Lambda$, the form factor deviates from the form factors obtained at higher $\Lambda$ values. This is due to the fact that the cutoff is not sharp enough, leading to possible deviations in the numerical integration of the code. However, in general such deviations are not observed.

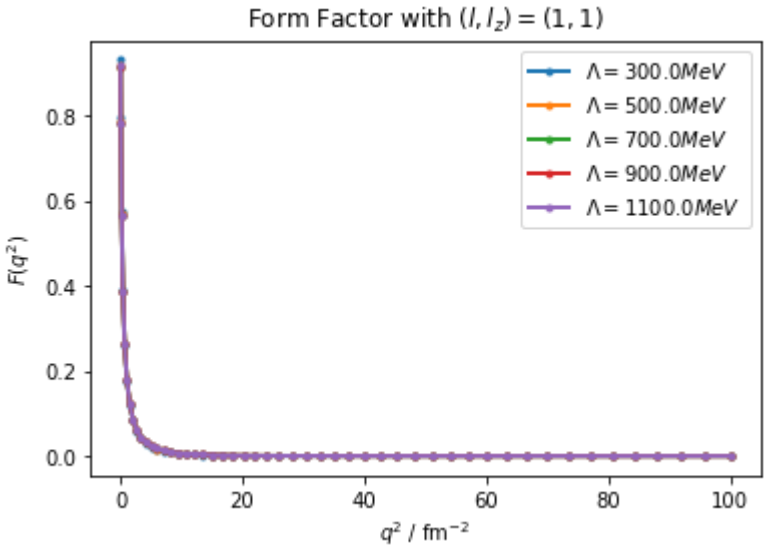
In general, we observe that for lower values of $\vec{q}$ the form factor behaves as $r^{-n}$ until some minimal value, in which it increases as $r^m$ slowly until it approaches zero. This is very reminiscent of the Leonard-Jonnes Potential observed for strong interactions.

For different values of $l, l_z$, however, we see different structures in the form factor, ex. if $(l, l_z) = (1, \pm 1)$ we obtain a $r^{-n}$ dependence for all values of $q$ instead.

In [22]:
```
# now plotting for differnet values of Lambda
Nq = 50
q_arr = np.linspace(0, 10, Nq)
nx_grid=40
l, lz = 1,1   # specific angular momentum state

F_arr_lmbda = np.zeros((len(res), Nq))

for i, (Lmbda, _) in enumerate(res[::2]):

    wf, pgrid, rms = output_arr[i]

    for j, q in enumerate(q_arr):
        pgrid_new, wf_new = interp_wf(wf, q, pgrid)

        theta, costheta_new = eval_costhetas(q, pgrid_new, pgrid)
```

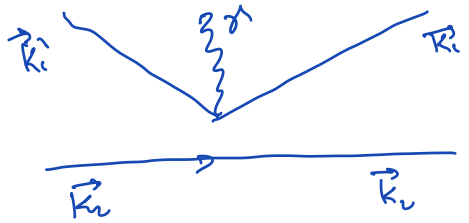
```
        fx = eval_Fx(q, l, lz, wf, wf_new, pgrid_new, pgrid)
        # played around values until we get F(0) = 1
        F_arr_lmbda[i,j] = eval_Fq2(fx, pa=0.55, pb=1., pc=20.0)
    # wf new
```

In [23]:
```
fig, ax = plt.subplots(figsize=(6,4))
for i, (Lmbda, _) in enumerate(res[::2]):
    ax.plot(q_arr**2., F_arr_lmbda[i,:], marker="o", ls="-", lw=2.0,
ax.set_xlabel("$q^2$ / fm$^{{-2}}$")
ax.set_ylabel("$F(q^2)$")
ax.set_title("Form Factor with $(l, l_z) = ({1}, {2})$".format(0, l,
ax.legend();
```




In [ ]:

1). The situation given is,




The value of $\vec{k}_1$ changes, but $\vec{k}_2$ does not.

Using the Jacobi coordinates, which are:

$$\vec{p} = \frac{1}{m_1+m_2}(m_2\vec{k}_1 - m_1\vec{k}_2)$$

$$\vec{p} = \vec{k}_1 + \vec{k}_2$$

$$\Rightarrow \quad \vec{p} = \frac{(m_2\vec{k}_1 - m_1(\vec{p}-\vec{k}_1))}{m_1+m_2} = \frac{-m_1\vec{p} + (m_2+m_1)\vec{k}_1}{m_1+m_2}$$

$$\vec{p} = \frac{-m_1\vec{p}}{m_1+m_2} + \vec{k}_1$$

$$\vec{k}_1 = \vec{p} + \frac{m_1\vec{p}}{m_1+m_2}$$

$$\vec{k}_2 = \vec{p} - \vec{k}_2 = \frac{m_2\vec{p}}{m_1+m_2} - \vec{p}$$

Before the interaction

$$\vec{P}' = \vec{k}_1' + \vec{k}_2$$

$$\vec{P}' - \vec{P} = \vec{k}_1' - \vec{k}_1$$

$$\vec{P}' = \frac{1}{m_1 + m_2}\left(m_2\vec{k}_1' - m_1\vec{k}_2\right)$$

$$\vec{P}' - \vec{P} = \frac{m_2}{m_1+m_2}\left(\vec{k}_1' - \vec{k}_1\right)$$

$$\vec{P}' - \vec{P} = \frac{m_2}{m_1+m_2}\left(\vec{P}' - \vec{P}\right)$$

Given, $\langle \vec{k}_1' | \rho(\vec{q}_n) | \vec{k}_1 \rangle = \delta(\vec{k}_1' - \vec{q}_n - \vec{k}_1)$

(As $\vec{P}' - \vec{P} = \vec{k}_1' - \vec{k}_1$)

$$\langle \vec{k}_1' | \rho(\vec{q}_n) | \vec{k}_1 \rangle = \delta(\vec{P}' - \vec{q}_n - \vec{P})$$

$$= \langle \vec{P}' | \rho(\vec{q}_n) | \vec{P} \rangle$$

$\Rightarrow$ We can then evaluate,

$$\langle \Psi\vec{P}' | \rho(\vec{q}_n) | \Psi\vec{P} \rangle = \langle \Psi(\vec{P}') | \Psi(\vec{P}) \rangle \langle \vec{P}' | \rho(\vec{q}_n) | \vec{P} \rangle$$

$$= \langle \Psi(\vec{P}') | \Psi(\vec{P}) \rangle \delta(\vec{P}' - \vec{P} - \vec{q}_n) \quad -(1)$$

e first term can be simplified:

$$\langle \Psi | \Psi \rangle = \langle \Psi(\vec{p}', \vec{P}') | \Psi(\vec{p}, \vec{P}) \rangle$$
$$= \langle \Psi(\vec{p}') \cdot \varphi(\vec{P}') | \Psi(\vec{p}) \varphi(\vec{P}) \rangle$$
$$= \langle \Psi(\vec{p}') | \Psi(\vec{p}) \rangle$$

Since, $\vec{q}_h = \vec{k}_1' - \vec{k}_1 = \vec{p}' - \vec{p}$

Taking identical particles, $m_2 = m_1$,

$$\vec{p}' - \vec{p} = \frac{m_2}{m_1+m_2}(\vec{p}_2' - \vec{p}) = \frac{1}{2}(\vec{p}' - \vec{p})$$

$$(\vec{p}' - \vec{p} = \vec{q}_h)$$

$$\Rightarrow \quad \vec{p}' = \vec{p} + \frac{1}{2}\vec{q}_h \Rightarrow \vec{p} = \vec{p}' - \frac{1}{2}\vec{q}_h$$

Using this: $\langle \Psi | \Psi \rangle = \langle \Psi(p) | \Psi(p) \rangle$

$$= \langle \Psi(\vec{p}) | \Psi(\vec{p}' - \tfrac{1}{2}\vec{q}_h) \rangle$$

In integral notation,

$$\langle \Psi(\vec{p}') | \Psi(\vec{p} - \tfrac{1}{2}\vec{q}_h) \rangle = \int d^3p' \, \Psi^*(\vec{p}') \Psi(\vec{p} - \tfrac{1}{2}\vec{q}_h)$$

Putting this back in ① we have:

$$\langle \Psi \vec{p}' | \rho(\vec{q}_h) | \Psi \vec{p} \rangle = \delta(\vec{p}' - \vec{q}_h - \vec{p}) \int d^3p' \, \Psi^*(\vec{p}') \, \Psi(\vec{p} - \tfrac{\vec{q}_h}{2})$$

So,

$$\boxed{F(\vec{q}_h^{\,2}) = \int d^3p' \, \Psi^*(\vec{p}') \, \Psi(\vec{p}' - \tfrac{\vec{q}_h}{2})}$$

2). We have,

$$F(\vec{q}_n^{\,2}) = \int d^3p' \; \Psi^*(\vec{p}') \; \Psi(\vec{p}' - \tfrac{1}{2}\vec{q}_n)$$

The wave function can be decomposed using Partial waves, using spherical harmonics.

$$\Psi(\vec{p}') = \sum_{lm} \Psi_{lm}(p') \, Y_{lm}(\hat{p}') = \sum_{lm} \Psi_l(p') \, Y_{lm}(\hat{p}')$$

Substituting this, we get:

$$F(\vec{q}_n^{\,2}) = \sum_{l',m'} \sum_{l,m} \int d^3p' \; \Psi^*_{l'm'}(p') \, Y^*_{l'm'}(\hat{p}') \, \Psi_{lm}(|\vec{p}' - \tfrac{1}{2}\vec{q}_n|) \, Y_{lm}(\widehat{\vec{p}' - \tfrac{1}{2}\vec{q}_n})$$

In spherical coordinates in momentum space, the volume element $d^3p'$ can be written as:

$$d^3p' = p'^2 dp' \sin\theta \, d\theta \, d\phi$$

None of the parameters inside the integral depend on $\phi'$. So, integrating over $d\phi'$ can be separated out:

$$\int_0^{2\pi} d\phi = 2\pi$$

It is also given that, only the $l=2$ partial wave contributes, as $\vec{q}_n = q\hat{z}_2$. So, we get:

$$F(\vec{q}'^2) = 2\pi \int dp' p'^2 \int_0^{\pi} d\theta \sin\theta \cdot \Psi^*_{l_2}(p') Y^*_{l_2}(\hat{p}') \times \Psi_{l_2}\left(\left|\vec{p}' - \tfrac{1}{2}\vec{q}\right|\right) Y_{l_2}\left(\overrightarrow{\vec{p}' - \tfrac{1}{2}\vec{q}}\right)$$

We can write, $x = \cos\theta$.

$dx = -\sin\theta \, d\theta$.

limits would change to:

$\theta : 0$ to $\pi$ , $x : 1$ to $-1$

Using the '-' sign in $dx$, we can change the limits to,

$x : -1$ to $1$.

Doing this,

$$F(\vec{q}'^2) = 2\pi \int dp' p'^2 \int_{-1}^{1} dx \, \Psi^*_{l_2}(p') Y^*_{l_2}(\hat{p}') \Psi_{l_2}\left(\left|\vec{p}' - \tfrac{1}{2}\vec{q}\right|\right) Y_{l_2}\left(\overrightarrow{\vec{p}' - \tfrac{1}{2}\vec{q}}\right)$$

5). The form factor is defined as

$$F(\vec{q}_h^{\,2}) = \int d^3p'\, \Psi^*(\vec{p}') \, \Psi(\vec{p}' - \tfrac{1}{2}\vec{q}_h)$$

when, $\vec{q}_h^{\,2} = 0$,

$$F(0) = \int d^3p\, \Psi^*(\vec{p}') \, \Psi(\vec{p}') = 1.$$

[As, $\Psi(\vec{p})$ is normalised]

So, $\boxed{F(0) = 1}$

$$\Rightarrow \frac{\partial F(\vec{q}_h^{\,2})}{\partial \vec{q}_h^{\,2}} = \frac{\partial}{\partial \vec{q}_h^{\,2}} \int d^3p\, \Psi^*(\vec{p}') \Psi\left(\vec{p} - \frac{\vec{q}_h}{2}\right)$$

Writing it in terms of the partial waves.

$$\frac{\partial F}{\partial \vec{q}_h^{\,2}} = \frac{\partial}{\partial \vec{q}_h^{\,2}} \left[ 2\pi \int dp' p'^2 \int_{-1}^{1} dx\, Y^*_{l_2}(\hat{p}') \, Y_{l_2}(\vec{p}' - \tfrac{1}{2}\vec{q}_h) \, \Psi^*_{l_2}(p') \, \Psi_{l_2}(|\vec{p}' - \tfrac{q_h}{2}|) \right.$$

$$|\vec{p}' - \frac{\vec{q}_h}{2}| = \sqrt{p'^2 + \frac{\vec{q}_h^{\,2}}{4} - |\vec{p}|\,|\frac{\vec{q}_h}{2}|\, x} \qquad (x = \cos\theta)$$

$$(|\vec{q}_h| = \sqrt{\vec{q}_h^{\,2}}.)$$

$$|\vec{p} - \vec{q}_h| = \sqrt{\vec{p}^{\,2} + \vec{q}_h^{\,2} - |\vec{p}|\,|\vec{q}_h|\, x}$$

Only the last term would have a dependence on $\bar{q}_h^2$

$$= \left( \bar{p} \cdot \frac{1}{2}\bar{q}_h \Rightarrow \cos\theta = \frac{\bar{p}\cdot\bar{q}_h/2}{|\bar{p}|\,|\bar{q}_h|} = \frac{\bar{p}\cdot\bar{q}_h}{|\bar{p}|\sqrt{q_h^2}} \right)$$

$$\Rightarrow \frac{\partial \Psi(|\bar{p} - \bar{q}_h|_2|)}{\partial \bar{q}_h^2} = \frac{\partial \Psi}{\partial q_h^2} \times \frac{\partial}{\partial}$$

$$\langle r^2 \rangle = \int d^3p \, \Psi^*(\vec{p}) \, \hbar^2 \left( \frac{\partial^2}{\partial p^2} \Psi(\vec{p}) \right)$$

RHS

$$\text{LHS}: \quad \int d^3p \, \Psi^*(p) \frac{\partial}{\partial \bar{q}_h^2} \left( \Psi(\bar{p} - \frac{\bar{q}_h}{2}) \right)$$

$$-\hbar^2 \frac{\partial^2 \Psi}{\partial p^2} = \frac{\partial}{\partial q^2} \cdot \left( \Psi\left( \vec{p} - \tfrac{1}{2}\vec{q} \right) \right) = \frac{\partial p}{\partial q^2} \cdot \frac{\partial}{\partial p} \cdot \left( \Psi\left( \underbrace{\vec{p} - \tfrac{1}{2}\vec{q}}_{\Lambda} \right) \right)$$

q q

we should show that these two would match