PHAS0030: Computational Physics
Session 9: Integral transforms

David Bowler

In this session, we will consider the representation of functions and data in different ways. We will consider polynomial representations and how they can be used for interpolation, and trigonometric representations such as the Fourier transform. We will examine Fourier transforms and their applications in some detail, and briefly mention other approaches.

1 Objectives

The objectives of this session are to:

- Consider different ways of representing functions, including interpolation schemes
- Review the basic ideas behind Fourier transforms, the discrete form of the Fourier transform and efficient implementations
- Use Fourier transforms for different applications in one and two dimensions
- Briefly note other transforms

2 Review of Session 8

In the eighth session we considered stochastic processes and how they can be used in computational physics. We discussed different approaches to the creation of pseudo-random numbers on computers (paying attention to uncorrelated sequences) and modelled simple stochastic processes using these numbers. We investigated how to produce different probability distributions for random numbers, and introduced the Monte Carlo approach to integration, noting the importance of sampling. We introduced Monte Carlo simulation and applied it to a simple model of spins (the Ising model).

3 Representing functions

The representation of a function in terms of known, simple functions or value sampled at points in space or time is a central part of physics, and particularly computational physics. It allows us to find solutions for physical problems, and to calculate the properties of systems that we are modelling. We encountered this idea when solving differential equations via finite differences: the functions there were represented by their values at discrete points in space and time.

There are, of course, many ways to represent a function, $f(x)$; the most common include: values on a grid; in terms of powers of x, or polynomials in x (a Taylor series, in effect); and using trigonometric functions (we will consider these in detail in Sections 4 and 5). In quantum mechanics, you have learned about the Legendre polynomials, which allow us to represent an arbitrary angular momentum[1]. It is important to consider the domain within which you want to represent the function, as well as how to represent it.

Generally, we can write:

$$f(x) = \sum_n c_n \, b_n(x)$$

where the functions used for the representation are a set $b_n(x)$ and the coefficients are c_n. The number of functions will often be, in principle, infinite, but this is not useful in practical calculations, so we will need to truncate the expansion. Formally, we can calculate the coefficients using the following approach:

$$\begin{aligned}
\int dx f(x) b_m(x) &= \sum_n c_n \int dx b_n(x) b_m(x) \\
a_m &= \int dx f(x) b_m(x) \\
S_{nm} &= \int dx \, b_n(x) b_m(x) \\
a_m &= \sum_n c_n \, S_{nm} \\
\Rightarrow c_n &= \sum_m S^{-1}_{nm} \, a_m
\end{aligned}$$

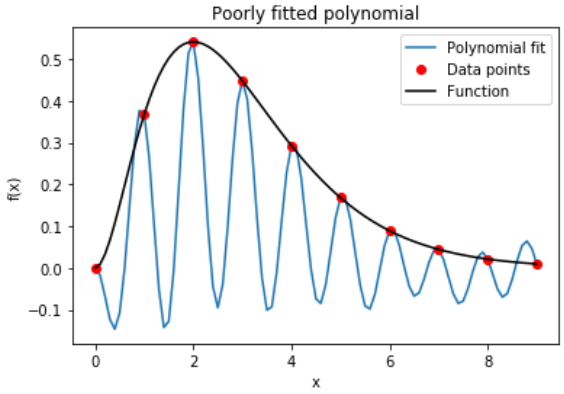
From these formulae, we can see that having an orthogonal basis (where $S_{mn} = 0$ if $m \neq n$) makes the calculation much simpler (indeed, for a large number of functions, the inversion of the matrix can be very time consuming[2]). Practically, however, we rarely find the coefficients in this way; normally we use discretely sampled data with numerical integrals and use numerical inversion techniques (or we can treat the problem of finding the coefficients as an optimisation, and use methods from Session 2 to find the coefficients).

Using a representation allows us both to solve for unknown functions (e.g. on a grid with a differential equation) and to find values of discretely samples functions or data between samples, via interpolation. But you should be aware that interpolation is an inherently

[1] More generally in quantum mechanics, we often use a *basis* set to represent the wavefunction; you will see how this leads to a matrix representation of quantum mechanics next year.

[2] Matrix inversion scales with the cube of the matrix size. Moreover, in the case of simple powers of x over the domain from 0 to 1, the matrix S becomes the Hilbert matrix, $S_{mn} = 1/(m + n - 1)$ which becomes almost impossible to invert *numerically* for a size larger than ten.

dangerous procedure[3], and can lead to significant errors if care is not taken; it is perfectly possible to create a polynomial which fits exactly to a set of data points but oscillates wildly in between the points, while the desired function is smooth, as shown below.



An example of over-fitting a polynomial to data points; here a Legendre polynomial of order 40 has been fitted to 10 data points.

An important question to consider is the domain over which fitting or interpolation is being performed: local, or global. A local interpolation will use a few points around the value of x where the function is needed; at the simplest level, using the points either side of x we can fit a straight line. One of the most common local fitting processes is the cubic spline: splines are piecewise polynomials (a different polynomial is used between different grid points) which match derivatives up to a certain order at data points. The cubic spline matches up to the second derivative.

A global fit (i.e. covering the whole domain in which there is data available) will use a family of functions, normally either polynomial or trigonometric, with the number of functions included in the expansion giving the order, N. The most common polynomials include Legendre polynomials and Chebyshev polynomials, which can be generated by recursion and are defined over a limited range (-1 to 1, which is easily re-scaled); in these cases, a least-squares fit is made to find the coefficients that produce a function that matches the data points as well as possible. It is vital that you never use an order higher than the number of data points (otherwise your problem will be under-determined: there will be fewer data points than coefficients).

It is almost always possible to find pre-written Scipy functions to perform the fitting necessary for interpolation (local or global), and this is often the most efficient approach. When using these functions it is important to check for any warnings or errors that are returned.

[3] Extrapolation is even more dangerous, and should only be attempted *in extremis*.

3.1 Exercises

3.1.1 In-class

1. Using `np.arange`, create an array of data points from 0 to 9 with spacing 1, and evaluate the function $f(x) = x^2 e^{-x}$ at those data points. Store the result in an array.

2. Import the `interpolate` module from scipy[4]. Use the function `interp1d` to fit a cubic spline to your data points (pass the x and $f(x)$ values from the previous question and the parameter `kind='cubic'`). Note that this returns a new function, say `f_interp(x)`, so you will end up with something like this:

   ```
   f_interp = interp1d(x_samp,y_samp,kind='cubic')
   ```

   where you can now call `f_interp(x)`. Plot the data points and the interpolated function using a finer grid of x points (spacing of 0.1 or smaller), and also plot the error in the interpolated function on a separate graph.

3. The function `np.polynomial.chebyshev.chebfit(x,y,order)` returns an array of coefficients for a Chebyshev polynomial, fit to the data points `x, y` (both are arrays). The array of coefficients, `coeff`, can be passed to the function `np.polynomial.chebyshev.chebval(x,coeff)` to find the value of the expansion at `x` (which can be a point or an array). Use this function to fit to the data points from question 1 coefficients) for orders 4 to 8, and make a plot showing the *difference* between the fit function and the real function. Compare to the result from question 2.

3.1.2 Further work

1. The function `np.polynomial.legendre.legval(x,c)` evaluates the value of a Legendre series at the point `x` given coefficients held in the array `c`. Calculate the integrals $\int_{-1}^{1} d\,x P_n(x) P_m(x)$ for $n, m = 0 \to 4$ to check for orthonormality (the array of coefficients should have all entries equal to zero except for the m^{th} or n^{th} entry which should be one). You will have to create an array of x values, evaluate the polynomials, and integrate (using whatever numerical method you choose). Check that you have converged the integrals with respect to grid spacing. I used `coeffs = np.eye(5)` to create a (5×5) identity matrix of coefficients, and passed the appropriate row of the matrix as the coefficients for each value of n or m.

2. Plot the first five or ten Legendre polynomials for $-1 < x < 1$.

3. Test the orthogonality of the first five Chebyshev polynomials, where the required integral is:

$$\int_{-1}^{1} T_n(x) T_m(x) / \sqrt{1 - x^2}\, dx$$

[4] Use the command `from scipy import interpolate`.

Here $1/\sqrt{1-x^2}$ is used to define the inner product between the polynomials. You will need to be careful with the value of this factor at the limits. Use the function `np.polynomial.chebyshev.chebval(x,c)` and the same approach for coefficients as in the last question.

4. Plot the first five or ten Chebyshev polynomials for $-1 < x < 1$.

4 Fourier transforms: basics

The Fourier transform of a function $f(x)$ is defined as

$$g(k) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(x) e^{-ikx} dx$$

though the normalisation constant in front of the integral is sometimes set to 1 (with a constant of $1/2\pi$ applied to the inverse transform). We have used the pair k and x here, but ω and t are also transforms.

If we want to use this to solve problems on a computer, then we need to work with a discrete set of points; moreover, we need to work with a finite domain. If the function is periodic, creating a finite domain is easy; if it is not periodic, then we can still apply Fourier transforms by taking the part of the function that we are interested in (from $x = 0$ to $x = L$, say) and repeating it periodically[5]. We can then write:

$$F_n = \sum_{j=0}^{N-1} f(x_j) \exp\left(-2\pi i \frac{jn}{N}\right)$$

where we have taken N samples in the interval $0 \leq x < L$ specified by $x_j = jL/N$. We can then write the function as:

$$f(x_j) = \frac{1}{N} \sum_{n=0}^{N-1} F_n \exp\left(2\pi i \frac{nj}{N}\right)$$

This is the *discrete* Fourier transform (DFT); however, this simple implementation scales poorly (the effort goes as N^2, which should be clear from the definition, and as you will find in the exercises).

The fast Fourier transform (FFT) is described in many places (the original paper is here), so we will not discuss the details of the method here. The important point to note is that it scales as $N\log(N)$, and is therefore extremely efficient compared to the naive DFT. The data covers both positive and negative frequencies/wavevectors, and is arranged from zero to

[5] We do therefore have to throw away the parts of the function that lie outside that range, and impose an artificial periodicity.

the largest positive frequency and then from the largest negative frequency to the smallest negative frequency (smallest in the sense of magnitude).

The indexing is subtly different depending on whether you have an even or odd number of points:

- Even
  - From 0 to $k_{max} - dk$: `coeff[0:n/2]`
  - From $-k_{max}$ to $-dk$: `coeff[n/2+1:]`
- Odd
  - From 0 to k_{max}: `coeff[0:(n-1)/2]`
  - From $-k_{max}$ to $-dk$: `coeff[(n+1)2:]`

The spacing, $dk = 2\pi/L$, can be used to convert the index number of the coefficient into a wavevector or frequency. Note that if you are creating an array to store a function that will be periodic from $x = 0 \rightarrow L$, then you do *not* want to include L in the array for x (check that you understand why). In this situation, `np.arange` is very useful.

Functions that are localised in real space will be delocalised in Fourier space, and vice versa. One exercise in the further work demonstrates this with Gaussians. Sharp transitions in a function (e.g. a square wave) will lead to rapid oscillations in Fourier space, and it will be very difficult to represent a function with a sharp feature accurately with a Fourier transform[6].

Thinking about the problem from the other side, we can use a Fourier transform as a filter: to remove frequencies below or above a particular limit; or to remove noise, which may be seen at different frequencies with small amplitudes. Careful plotting of the data will enable us to see where the noise is likely to be, and how best to remove it. This type of approach, spectral analysis, is very powerful.

4.1 Exercises

4.1.1 In-class

1. Write a function to implement the DFT using the formula given above. When you create the array to store the coefficients, remember that it needs to be complex.

2. Now create a simple sine function, $\sin(kx)$ for $0 < x < 2$ and with $k = \pi$, in an array of length 100 (use `np.arange` and an appropriate step size) and use your function to transform it. Plot the coefficients (use `abs`) to be sure that it is correct.

[6] In fact, there is tendency for a Fourier expansion to overshoot the original function at a sharp transition; this is known as the Gibbs phenomenon.

3. Now do the same for a longer array (I suggest length 1,000 and strongly advise against going much beyond this) and estimate roughly the time required to run. Use `np.fft.fft` to do the same calculation with a fast Fourier transform, and compare the times.

4. Create an array containing a saw-tooth wave, running from -0.5 to 0.5 as x runs from 0 to 1; include two periods in your array, which should have at least 100 entries. Use `np.fft.rfft` (an FFT for real data) to create the Fourier transform of your wave and plot it. (You should plot the *absolute* value of the Fourier transform - remember that the coefficients will be complex.)

   Note that you can make the first period of the saw-tooth using $y = x - 0.5$ for $0 \leq x < 1$; you can then create a second period either by using something like `np.append` or by writing $y = x - 0.5$ for $0 \leq x < 2$, and applying $y = y - 1$ for $1 \leq x < 2$.

5. Now explore what happens when you build an approximation to the saw-tooth wave by including on the first n terms from the full Fourier transform. (I set up an array of zeros with the same length as the Fourier transform and copied the first 10, 20 and 30 elements respectively.) Use `np.fft.irfft` for the inverse of the real Fourier transform to transform these back, and plot them along with the original wave.

6. Create a signal with some noise using the following command: `np.sin(xnoi) + np.cos(5*xnoi) + 0.2*np.random.random(len(xnoi))` where `xnoi` is an array from 0 to 2π (feel free to experiment with other forms of noise; we will look at another in the further work). Plot your signal. Now Fourier transform using `np.fft.fft` and plot the result: notice how there are only two frequencies which are strong (i.e. have large coefficients). Using `np.where`, set all coefficients below a threshold to zero (use your judgement to set the threshold) and Fourier transform back using `np.fft.ifft`. Plot the result and compare to the original signal without noise.

4.1.2 Further work

1. For a grid defined by $-5 < x < 5$ with a spacing of 0.1, create a Gaussian with $\sigma = 1.0$ and Fourier transform it. To plot the Fourier transform, you can use the routines `np.fft.fftfreq(n,dx)` to return the frequencies and `np.fft.fftshift(coeffs)` to shift the coefficients so that the zero frequency term is in the middle. (You could also just copy the second half of the array of coefficients to the start of a new array and copy the first half of the array of coefficients to set second half of the new array[7].) Do the same calculation for Gaussians with $\sigma = 0.5$ and $\sigma = 2.0$ and make two plots: one in real space, one in Fourier space. Notice how the width of the functions and their transforms relate to each other.

2. We will repeat the noise calculation in question 6 above, but this time add `np.random.random(len(xnoi)) * np.cos(6*xnoi)` which is harder to extract. After

[7] This is simpler than it sounds: `Fplot[0:int(n/2)] = F[int(n/2):]` and `Fplot[int(n/2):] = F[0:int(n/2)]`

Fourier transforming, plot the coefficients for the first 10 frequencies and notice where they are large. Filter the coefficients by setting a threshold and setting all coefficients below the threshold to zero using `np.where`, transform back and plot. Experiment with the effect of the threshold.

3. This time, filter by setting all coefficients for frequencies above a certain magnitude to zero (you can do this using the index of the coefficient array). Remember that you will need to leave the low positive *and* negative frequencies unchanged (you need to set the middle of the array to zero; use an array slice like `a[n:-n]` to do this, though the upper limit may need to be moved by one if you have an odd number of points).

5 Fourier transforms: more detail

Fourier transforms can be very useful when applied to differential equations, both analytically and computationally; in particular, for a periodic function $f(x)$ with Fourier transform $g(k)$, the Fourier transform of the *differential*, df/dx is simply $ikg(k)$. This gives us a particularly easy way to solve equations like Poisson's equation:

$$\begin{aligned} \nabla^2\phi &= -\rho/\epsilon_0 \\ -k^2F[\phi] &= -F[\rho]/\epsilon_0 \end{aligned}$$

where $F[]$ indicates a Fourier transform of a function. We can solve Poisson's equation for a periodic charge density simply by Fourier transforming the charge density, dividing by the square of the wavevector, and performing the inverse Fourier transform[8] Considerable care needs to be used with systems of charges which are periodic: if the overall charge is not zero, then the energy becomes infinite (though this is true for any approach to finding the potential). Moreover, the potential is long-ranged and using this method to calculate the potential for non-periodic systems forced to be periodic can be slow to converge.

We should also mention the concept of aliasing. Data which is sampled at a particular rate, with spacing Δ between samples, can be represented with a Fourier transform containing frequencies up to a critical frequency, $1/2\Delta$ (known as the Nyquist frequency). If the data contains frequencies above this, then the information in this frequency range will be moved below the critical frequency. This is a phenomenon known as aliasing; so long as you are careful about the sampling rate used relative to the problem you are working on, it should not be a problem.

5.1 Other transforms

There are other integral transforms that are used. The Laplace transform is common analytically, but not widely used computationally. There is a field of transforms known as *wavelet* transforms which have similar applications in image processing to Fourier

[8] There is of course a problem when $k^2 = 0$; however, for a charge-neutral system the coefficient at that point is also zero and the singularity is avoided.

transforms, but use a different class of functions for the transforms. Wavelets are maximally localised in both real and transformed space (where Fourier transforms use functions that are completely delocalised in real space and completely local in transform space). However, we will not discuss these further.

It is also worth mentioning the Hankel or Bessel transform, which is extremely useful for transforming functions with circular or spherical symmetry. In these cases, working in polar coordinates, the angular terms can be found analytically, while the radial term changes to involve integrals with Bessel functions. This reduces the 2D or 3D problem to a one dimensional one, which can give considerable improvements in accuracy and computational cost.

5.2 Exercises

The final two questions might be considered further work; it may be possible to get to the end of question 3 in class, but question 4 is a little more challenging.

1. We will use the Fourier transform of the sine function from question 2 in Section 4 to test differentiation. Create an array of k values using the recipe in Section 4 (running from 0 to k_{max} and then $-k_{max}$ to $-dk$ but paying attention to the number of points), and make a new array which is $ikF[\sin(kx)]$. Perform an inverse Fourier transform using `np.fft.ifft` and plot it alongside $k\cos(kx)$. Check that they match.

2. Now we will consider a 2D charge density and the equivalent potential. Create 2D arrays of x and y with a side length $L = 40$ with 100 points along a side, using `np.meshgrid`. With a value $k = 2\pi/L$ make a 2D charge density $\sin(kx) \times \sin(ky)$ and plot it with `plt.imshow`.

3. Now Fourier transform the density using `np.fft.fftn`, the N-dimensional generalisation of the FFT (you just need to pass the 2D array). Create 2D arrays containing k_x and k_y for the reciprocal space grid (you will need to use `np.meshgrid` again) and calculate $F[\rho]/k^2$. To avoid a warning about division by zero, you could add a very small value (I used `1e-16`) to k^2 and set the `[0,0]` element to zero. Fourier transform back (`np.fft.ifftn`) and plot. I found it helpful to use both `plt.imshow` and `plt.contour`.

4. The potential can be calculated using the explicit formula:

$$V(\mathbf{r}) = \int d\mathbf{r} \frac{n(\mathbf{r}')}{|\mathbf{r} - \mathbf{r}'|},$$

5. but we have to think carefully about boundary conditions. The simplest approach is just to use the density from the previous question; we will do this here. (We could implement periodic boundary conditions, but that would require some work; the next question shows an alternative approach.)

   Now loop over all points in the grid using `enumerate` for both x and y, and for each point $\mathbf{r}$ on the grid calculate a 2D array containing $|\mathbf{r} - \mathbf{r}'|$ for *all* other grid points, $\mathbf{r}'$

(this is straightforward using Numpy arrays: you can find the x and y components just by subtracting and then calculate the distance using `np.sqrt`).

Use this array to find $n(\mathbf{r}')/|\mathbf{r}-\mathbf{r}'|$ at every point on the grid (again, very easily using Numpy arrays) and integrate over the grid to get the potential $V(\mathbf{r})$ (I just summed and scaled by the grid spacing squared). You will need to be careful about the point where $\mathbf{r}=\mathbf{r}'$; the contribution to the potential can be taken to be zero. Plot the potential, and compare it to the answer from the previous section.

6. Another approach is to use a system two or three times larger than before and take only the central part of the potential. Create a grid twice the size of the one in the previous question (I used 200 points which is not too intensive), and create a charge density on this grid using the same formulae as above. Now loop over all points in the grid using `enumerate`, and for each point $\mathbf{r}$ on the grid evaluate $|\mathbf{r}-\mathbf{r}'|$ for *all* other grid points (this is trivial using numpy arrays). Use this to find $n(\mathbf{r}')/|\mathbf{r}-\mathbf{r}'|$ and integrate to get the potential (I just summed and scaled by the grid spacing squared). You will need to be careful about the point where $\mathbf{r}=\mathbf{r}'$. Plot the central part of the potential (with 200 points you can use the array slice 50:150) and compare to the result using FFTs.

7. As a final extension, if you wanted to solve for the first charge density using FFTs without making the density itself periodic, you could place the density into a larger grid, setting all entries to zero outside the central part. How large does the zero-padding area have to be before the potential compares well to Q4?

6 Assignment

Note that this assignment is *not* marked. You will apply fast Fourier transforms (FFTs) to the output from the Session 8 assignment, which is provided as a file, `spins.txt`. We will use FFTs to differentiate and find the edges of the domains in the spin distribution.

1. Load the spins, using `spins = np.loadtxt('spins.txt')` or something similar. Plot with `plt.imshow` to be sure that you know what your starting distribution looks like.

2. Using `np.fft.fftn`, perform a Fourier transform of the spins in 2D. Plot the coefficients (note that you can shift the output array from the FFT so that the zero wavelength is in the centre of the plot using the command `np.roll(np.abs(tmp),(25,25),(0,1))`, which assumes an array size of 50, as given in the file). Plot the coefficients for an FFTs of an array of *random* spins in a second image (use the approach from Session 8 to generate the random array). Comment briefly on the differences between the two, thinking about where the largest Fourier components are located, and how this compares to the structures of the spins.

3. Now calculate the k-vectors (use lines like `k2darr[0:nk2dmax] = dk2d*np.arange(0,boxlen/2)` and `k2darr[nk2dmax:] = dk2d*np.arange(-boxlen/2,0)` after defining `dk2d`; then use `np.meshgrid` to generate an appropriate set of 2D vectors). Use these vectors to calculate the x and y derivatives of the array (scale

the FFT by ik_x or ik_y and apply the inverse FFT). Create an array $\sqrt{(\partial f/\partial x)^2 + (\partial f/\partial y)^2}$ and plot it. Taking a derivative like this is one approach to detecting edges in images; comment briefly on how effective you think that it is.

4. We can take a second derivative by scaling the Fourier transform by k^2 and reversing the transform. Do this and plot the magnitude of the result, alongside the spins and the first derivative from the previous question. Comment on whether you think the first or second derivative is a better approach for finding edges.

5. Finally, define an array, say `a`, in real space (the same size as your spin array) with zeros everywhere except: `a[0,0] = -1` and `a[0,1], a[0,-1], a[1,0], a[-1,0]` all set to 0.25 (you may recognise this as the finite difference 'stencil' for the second derivative). Take the Fourier transform of this array and plot it with `plt.imshow` (you don't need to shift it). Now multiply the Fourier transform of the array by the Fourier transform of the spins. Perform the inverse Fourier transform and plot the magnitude. Comment on how this relates to your answer from the previous question and why you think that this relationship exists.

7 Progress Review

Once you have finished *all the material* associated with this session (both in-class and extra material), you should be able to:

- Interpolate between data points safely and accurately
- Understand how to use Fourier transforms
- Filter data to remove noise or otherwise adapt
- Calculate differentials of periodic functions using Fourier transforms