Documentation for Differential Forms Summer Project

Moustafa Gharamti, Samuel Kirwin-Jones, Maciej Tomasz Jarema

0.1 Geometry of stack vectors

0.1.1 Translations needed to define stack vectors and arrowheads

Stack vectors are covariant vectors, defined by planar sheets (lines, when working in 2D) perpendicular to arrows of the contravariant vector. The density of these planes, is determined by magnitude of the vector field at each point in space. These stack vectors, being covariant, correspond to differential forms, while arrow vectors (being contravariant) correspond to vector fields.
In python, these stack sheets have to be defined from the magnitude and direction of the input, based on x and y components of the 1-from.

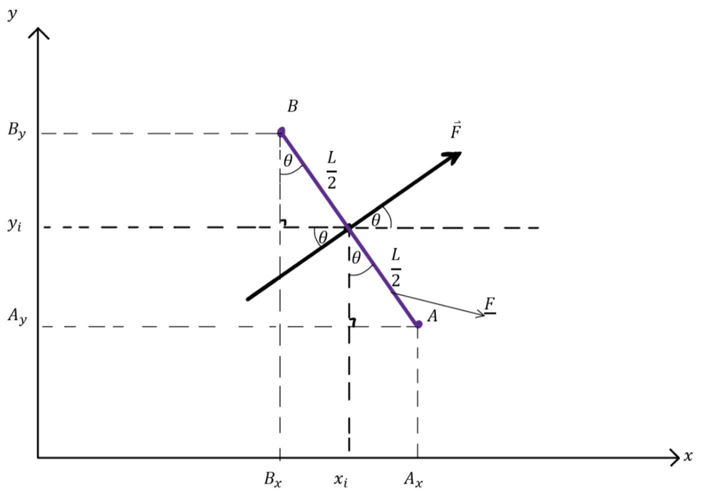

Figure 1: Sketch presenting the Geometrical arguments needed to define stack sheets, based on field magnitude and direction(indicated well by an arrow vector as shown)

For x_i and y_i marking the i^{th} considered position in the field, with its corresponding angle to the x-axis (ccw). Technically, points A and B as well as θ and $\vec{F}$ depend on the position in space that is considered, therefore, should include the subscript, 'i'. This was not added for figure clarity.
The stack, shown in purple, is perpendicular to $\vec{F}$ with end points A and B separated by distance L (in the code, defined as a fraction of graph scale).
From these, through simple geometry, one obtains the following equations:

$$\begin{aligned} A_x &= x + \left(\frac{L}{2}\right)\sin(\theta)\,, \\ A_y &= y - \left(\frac{L}{2}\right)\cos(\theta)\,, \\ B_x &= x - \left(\frac{L}{2}\right)\sin(\theta)\,, \\ B_y &= y + \left(\frac{L}{2}\right)\cos(\theta)\,, \end{aligned} \tag{1}$$

describing the positions of points A and B in terms of their Cartesian components.
These also function generally as operations that displace a point on the vector in the direction perpendicular to the arrow, by a corresponding length - here by $\frac{L}{2}$.

To then displace the stack sheet by distance d in the direction parallel to the arrow as shown in Figure 2,

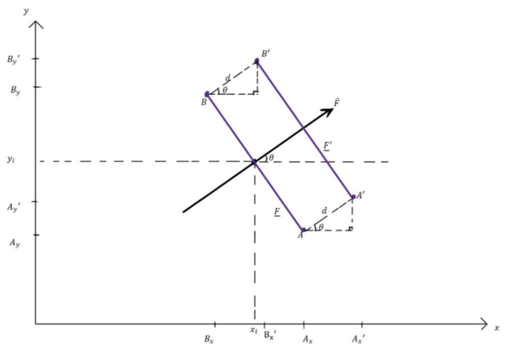

Figure 2: Sketch presenting the geometrical arguments needed to displace stack sheets in the direction parallel to the vector field direction at that point, (again - represented by an arrow)

we use the following translation equations:

$$\begin{aligned} A'_x &= A_x + d\cos(\theta)\,, \\ A'_y &= A_y + d\sin(\theta)\,, \\ B'_x &= B_x + d\cos(\theta)\,, \\ B'_y &= B_y + d\sin(\theta)\,, \end{aligned} \tag{2}$$

which again function as general operations for such parallel displacements by any distance d, from the centre.
To define the arrowhead on that stack vector, both translation operations, eqs. (1, 2), need to be used to obatin points to be connected as shown on the figure below

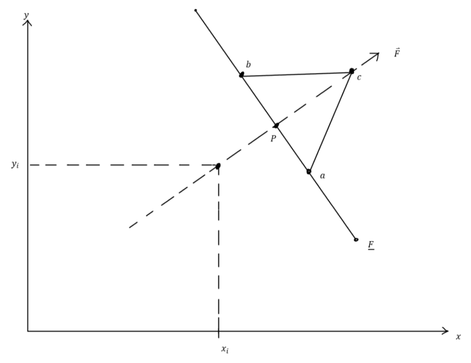

Figure 3: Sketch presenting the geometrical arguments needed to define the arrowhead on each stack vector)

$$\begin{aligned} P_i &= \left(x_i + \frac{L_s}{2}\cos(\theta_i),\ y_i + \frac{L_s}{2}\sin(\theta_i)\right) \\ a_i &= \left(P_{x,i} + \frac{L_s}{w_{head}}\sin(\theta_i),\ P_{y,i} - \frac{L_s}{w_{head}}\cos(\theta_i)\right) \\ b_i &= \left(P_{x,i} - \frac{L_s}{w_{head}}\sin(\theta_i),\ P_{y,i} + \frac{L_s}{w_{head}}\cos(\theta_i)\right) \\ c_i &= \left(x_i + \frac{L_s}{h_{head}}\cos(\theta_i),\ y_i + \frac{L_s}{h_{head}}\sin(\theta_i)\right) \end{aligned}$$

In these equations, variables are defined as shown on the above figure. L_s is the maximum thickness of the stack, parallel to the direction of the considered vector field, w_{head} is the width of the base of the arrowhead that rests on the last stack, and h_{head} is the height of the arrowhead parallel to the vector field direction.

Note that each has also been assigned a subscript i which does not appear on the figure. This is only done as these coordinates depend on the point in space that is currently being considered. Also note, that for a single sheet in the stack, the initial displacement from $(x,\ y)$ to P must be omitted.

0.1.2 General from of stack sheet positions, for any number of sheets

In the code, it was important to define the displacements of each stack sheet, form the middle position of the arrow vector, at each point in the field. Positioning of sheets depends on the magnitude of the vector field at that point in space. The larger the magnitude, the higher the sheet density. As the spread of the sheets is limited to a pre-defined maximum (on the plot, not physically), this increase in density corresponds to more stack sheets being present.
One way of plotting these, such that all sheets are equally-spaced is to consider odd and even number of stacks separately. We define them by displacing stack sheets along the arrow by certain lengths (as per the equations above) from the middle position (the considered position in the field).

For **odd number of stack sheets**, the following pattern was noticed:
For 1 sheet: no displacing is needed $\Rightarrow \pm 0$
For 3 sheets: none, and ends of the stack $\Rightarrow \pm 0$ and $\pm\frac{1}{2} \longrightarrow$ technically $\pm\frac{1}{2}\cdot L_s$
For 5 sheets: none, ends, and points equally between $\Rightarrow \pm 0$, $\pm\frac{1}{2}$ and $\pm\frac{1}{4}$
For 7 sheets: $\Rightarrow \pm 0$, $\pm\frac{1}{2}$, $\pm\frac{2}{6}$ and $\pm\frac{1}{6} \longrightarrow$ At this point, note that $\frac{3}{6}=\frac{1}{2}$ (obvious but important)
For 9 sheets: $\Rightarrow \pm 0$, $\pm\frac{1}{2}$, $\pm\frac{3}{8}$, $\pm\frac{1}{4}$ and $\pm\frac{1}{8} \longrightarrow$ Again, $\frac{1}{4}=\frac{2}{8}$ and $\frac{1}{2}=\frac{4}{8}$
For 11 sheets: $\Rightarrow \pm 0$, $\pm\frac{1}{2}$, $\pm\frac{4}{10}$, $\pm\frac{3}{10}$, $\pm\frac{2}{10}$ and $\pm\frac{1}{10}$
$\vdots$
These display the following recursion:
The displacement along the arrow as a fraction of total stack length L_s, of the s^{th} sheet, when drawing n stack sheets overall, for odd n is:

$$\pm\frac{s}{(n-1)}\,,$$

where we require the fractional displacement (from the middle position) to not exceed half of the total length therefore:

$$1 < s < \frac{1}{2}\,(n-1)\ .$$

For **even number of stack sheets** this pattern emerges:
2 sheets:$\Rightarrow \pm\frac{1}{2}$
4 sheets:$\Rightarrow \pm\frac{1}{2}$ and $\pm\frac{1}{6}$
6 sheets:$\Rightarrow \pm\frac{1}{2}$, $\pm\frac{3}{10}$ and $\pm\frac{1}{10} \longrightarrow$ Note again that $\frac{1}{2}=\frac{5}{10}$.
From here on continue with those not reduced fractions:
8 sheets: $\Rightarrow \pm\frac{1}{14}$, $\pm\frac{3}{14}$, $\pm\frac{5}{14}$ and $\pm\frac{7}{14}$
10 sheets: $\Rightarrow \pm\frac{1}{18}$, $\pm\frac{3}{18}$, $\pm\frac{5}{18}$, $\pm\frac{7}{18}$ and $\pm\frac{9}{18}$
$\vdots$
These display the following recursion:
The displacement along the arrow as a fraction of total stack length of the s^{th} sheet, when drawing n stack sheets overall, for even n is:

$$\pm \frac{2s+1}{2\,(n-1)}\,,$$

where we require the fractional displacement to not exceed half of the total length therefore:

$$1 < s < \frac{1}{2}\,(n-2)\ .$$

Alternatively, each can be defined separately, manually, if a small amount of sheets is needed.

0.1.3 variables in stack plot code

This has been implemented in code. The parameters for this are as follows:

- L is the length of the positive x and y axes,
- pt_den is the number of points along each axis,
- a is a linear scaling of the field,
- u is the vector field $x-$component, v is the vector field $y-$component $\longrightarrow$ Alternatively: Fr is the radial component and $Ftheta$ is the angular component
- orientation is a sting that defines how the arrows pivot,
- scale is a linear scale on the quiver plot arrows,
- delta is the extra length along the axis to show, past the defined grid, full emerging arrows from border points,
- fract is the fraction of graph length equal to stack sheet in direction perpendicular to arrow,
- s_max is the maximum number of stacks to use,
- sheet_L is the length of stack perp. to arrow,
- s_L is the maximum length of stack sheet parallel to arrow,
- w_head is the width of the arrowhead base as the fraction of the stack sheet length perpendicular to the arrow,
- h_head is the length of the arrowhead parallel to the arrow, as the fraction of the total stack size parallel to the arrow,

0.2 Initial GUI of the quiver and stack plot

0.2.1 Explaining user defined functions in the GUI code

parity: this function takes an input of an integer and returns True (1) is it is even and False (0) when it is odd. It is useful when defining stack sheets as displacements from the middle position, as the formulas are different for even and odd number of sheets per stack.
G: it takes three inputs $\Rightarrow$ s: the recursion of sheet displacing from the middle position (which pair is being completed), n: how many sheets are to plotted in total and c: which is the **bool** value from the parity function
stack_plot: takes the following inputs:

- xg and yg: the grid of points to be used when plotting the field

- ax: the axis to plot on
- u and v: the $x-$ and $y-$components of the vector field to be plotted
- s_max: maximum number of sheets to plot, changes how dense the plot appears
- L: changes the size of axis (in both x and y equally, from origin)
- pt_den: defines the number of points on each axis that create the grid
- fract: defines the size of the stack sheet (as a square) as a fraction of total graph size
- arrows (optional, default=True): Bool variable, defines if arrows should be plotted on top of the stacks (when True), or if only stacks are to be plotted (when False)
- orientation (optional, default='mid'): sets the pivoting point of the arrows about that grid point that they are defined at.
- scale (optional, default=1): Linearly scales the arrows in the quiver plot
- w_head (optional, default=8): sets the fraction that defines the width of the arrowhead at its base (on the stack), from the total size of the stack.
- h_head (optional, default=4): Sets the fraction that defines the height of the arrowhead parallel to the vector field magnitude at that point, from the total size of the stack.

on_key_press: Function that tracks mouse key presses, needed for the 'Matplotlib' toolbar to function
format_eq: Takes a single string, converts all variables in it that are common in vector field equations, and turns them into things that python can understand. Returns the corrected string
eq_to_comps: Takes the two strings given by the user (equations for the field in the x and y directions) as well as the x and y grids. Uses the above function (format_eq) to make the string 'python readable'. If one or more of the strings does not contain x or y, it defines an array of ones and multiplies by the given constant. This is done for the component to be over all points along the grid and for shapes to match. Otherwise, it evaluates the given equation and returns the vector field components u and v in the usual way.
vect_type_response: Responds to changes in Radio-buttons that set the type of field to be plotted (arrow, stacks or both). Takes in a value from the Radio-buttons corresponding to the chosen field type. It clear the current plot. Checks which button has been selected and uses ax.quiver and previously user defined stack_plot to create the updated graph. It then updates it on the GUI by using canvas.draw() for the canvas being defined on the main window, in its own frame. Returns no variables.
PLOT_response: Responds to the 'PLOT' button being pressed. Updates the axis scaling, point density, maximum number of sheets per stack, linear scaling ('a') and the new field components. Takes in no input. collects all needed variables by the '.get()' method of Tkinter objects. After running, it plots the new specified field, with the new parameters as a stack only plot and changes the status of the Radio-buttons to one again be - stacks only (therefore for tensor (same variable name as in VFA java code) to equal 0)
custom_btn_response: when the button called 'customise' is pressed, this function responds by opening a new ('optimisation settings') window, in which features can be customised. It includes entry boxes where the user can input new values for parameters such as 'fract', 'w_head' and 'h_head' (described previously). These are initially filled with current values, changing them and pressing 'SUBMIT ALL' updates the plot.
custom_submission: Responds to the 'SUBMIT ALL' in the new 'optimisation settings' window. When the button is pressed, the input values are globally saved. The current axis are cleared, a new plot is constructed as per the new specifications and it is displayed on the 'canvas' of the correct frame.

The updated plot is initially only a stack plot, therefore the radio-buttons are returned to the original position of 'stacks'. The new window is then closed.

0.3 2-forms from 1-forms, exterior derivative (on $\mathbb{R}^m$)

The code includes a script that calculates 2-forms from a given 1-form in a specified number of dimensions. It requires input of the 1-form components respectively to their elemental 1-form (in code: string_x, string_y, etc.), variable arrays (in code: x, y etc.) as well as grids (in code: xg, yg, etc.) of these and a list of symbols for each variable used in the equations (coords). The code, so far, has to be changed manually to include the change of all given components from **string** type to **sympy.core.mul.Mul**. The number of used dimensions has to also be input as an integer ('m' in the code). The variable, 'expressions' has to also be appended with the correct number of components.
Python cycles over the components and the given symbols for variables, differentiating each. To improve efficiency, the derivatives have not been completed when components are to be differentiated with respect to their corresponding elementary 1-form (these go to zero). The results are saved into an (m $\times$ m) array called 'ext_ds'. The array is of data type 'object', as it must store strings of arbitrary length in each of its components. It stores all the found derivatives, with components changing along the rows, and variables changing along the columns. Each derivative expression is also changed into a string, This is done to append needed prefixes onto them and to later use them with the eval() function, such as brackets and negatives. Minus sings (strings) are then appended to the upper right hand-side of the matrix. This is done to correct for the elementary 2-forms being in the wrong order.
This, completed, array is then taken through loops that extract the components of each elementary 2-form. These components are symmetrical elements, therefore extracting them includes taking elements with the same i and j, switched in the coordinates. A 'pair' variable is introduced to keep track of how many times the loop extracted a pair of 2-form components and merged them. As only 2-forms are being calculated, there is always 2 components being extracted toward one elementary 2 form (including zeros as components) The number that this variable reaches is the number of components of a 2 form on $\mathbb{R}^m$, which was found to be given by triangular numbers. Resulting pairs are stored in an array of 1 column and a number of rows determined by the triangular number of dimensionality. This array is called 'result', and stores objects (it must consist of strings of varying lengths for each element). Each time a new pair is being considered, the element from 'result' is cleared to exclude the initial 'NoneType' variable present. When appending the components, it is checked that if one or more of them evaluated to zero, they are not appended. The result is printed, as a vector, whose rows correspond to different 2-form elements. The order is as follows: For m=3:

$$dx \wedge dy,\ dx \wedge dz,\ dy \wedge dz.$$

For m=4 these become:

$$dx \wedge dy,\ dx \wedge dz,\ dy \wedge dz,\ dx \wedge dw,\ dy \wedge dw,\ dz \wedge dw,$$

etc.

These follow the component pairs as they appear in the lower-right of the matrix (ext_ds), in order that follows each line, until the main diagonal. It can be clearly visualised on the example of the 2-form on $\mathbb{R}^4$ as follows:

$$\begin{pmatrix} 0 & dy \wedge dx & dz \wedge dx & dw \wedge dx \\ dx \wedge dy & 0 & dz \wedge dy & dw \wedge dy \\ dx \wedge dz & dy \wedge dz & 0 & dw \wedge dz \\ dx \wedge dw & dy \wedge dw & dz \wedge dw & 0 \end{pmatrix} \tag{3}$$

Each component of the outcome ('result') is then formatted in such a way as to be understood by python's eval() function and the evaluation of each is saved into a variable called 'form_2'. This has to be composed of m dimensional arrays, one for each elemental 2-form component (given by the triangular numbers from the used dimensions ‘m’).

IMPORTANT - CONVENTION

There exists a convention on $\mathbb{R}^3$ which allows for simple matching of 2-forms to unit vectors used in vector calculus. By this convention, the elemental 2 forms used are not as stated above ($dx \wedge dy$, $dx \wedge dz$, $dy \wedge dz$), but instead, are given by: $dx \wedge dy$, $dz \wedge dx$, $dy \wedge dz$. This allows for use of the standard Cartesian coordinates with the simple rule that the 2 forms correspond to unit vectors. This convention was not followed in this code, as it is irrational to use it in $m > 3$, and it was established that it shadows an important aspect of differential forms. Wedge products are not defined in extra dimensions (in a way that a curl is defined in a direction the filed does not occupy, perpendicular to it). Wedge products have their orientation defined through the axial sense (clockwise and counter-clockwise). This means that their wedge product does not occupy extra dimensions. This can only be clearly retained, as used in higher dimensions, when the convention on $\mathbb{R}^3$ is abandoned. This is exactly what is done in the code. This code gives the 2 forms defined through clockwise (negative) and counter-clockwise (positive) orientations.

0.4 Inset Plots

0.4.1 Derivative Window

Selecting the ‘Derivative Plot’ radiobutton (top right frame) allows the user to display a small grid showing the local derivative of the field by clicking the vector field (top left frame) at the point of interest. The derivative field plots according to the ’tensor’ variable (i.e. if the vector field is currently plotting with stacks/arrows, the derivative field will also.)

The radiobutton variable ‘click_option’ is a tk IntVar and is converted to an integer in the response function ‘click_option_handler’. ‘click_option’ is then used to determine the required click control in the function ‘on_key_press’: if the ‘Derivative Plot’ radiobutton is selected, clicking calls ‘deriv_calc’ which creates the plots. The feature is disabled by selecting the ‘Tools’ radiobutton (‘Tools’ option refers to the improved functionality of the matplotlib toolbar i.e. for zooming or panning.) *Note: this method still needs work as the toolbar tools still function when ‘Tools’ is not selected.*

The ‘deriv_calc’ function takes the x, y data click coordinates, the pixel click coordinates to create an inset axis (‘deriv_inset_ax’) centred at the user’s selected point. The parameters which determine the axis properties are:

- dpd: point density of the derivative plot (user sets via drop down menu)
- d_length: size of the inset axis (user sets via drop down menu)
- d_range: x and y distance from the chosen plot point over which the derivative is taken (user varies using zoom slider)
- d_scale: scaling of the arrows in the derivative plot (user varies using zoom slider)

The plot is generated by creating new x and y meshgrids (dxg, dyg) with centres at the click data point. Previously mentioned ‘eq_to_comps’ is then called with the user defined components ‘string_x’ and ‘string_y’ to generate a local field ‘u1’ and ‘v1’. The derivative is taken by subtracting the components of the central grid element from every element in the array, leaving du1 and dv1. The ’if’ statement checks whether to plot with arrows/stacks/both. To plot the stacks, the ‘stack_plot’ function is called with changed input grids and axis to plot based on the internal parameters of the inset plot. The if/continue statement is entered this time, to prevent stacks plotting at the central grid position (as the derivative field is always zero there). Finally, the inset axis is displayed on the canvas using ‘canvas.draw’.

0.4.2 Zoom Window and Zoom Slider

The zoom window plots the local vector field components 'u1' and 'v1' on the 'dxg' and 'dyg' meshgrid arrays, in the same fashion as for the derivative. The zoom slider value is used in 'deriv_calc' to decrease 'd_range' and increase 'd_scale' such that the plot shown takes vectors from a much smaller region of the field and makes sure they are sized appropriately. The effect of this can be seen when zooming in on a region of the plot sufficiently, such that a constant field is displayed. As for the derivative feature, low magnifications no longer produce a viable representation of the derivative field, but upon zooming the user can see the derivative field emerge (which tends to a linear field at high magnification).

0.4.3 Divergence and Curl Windows

The divergence and curl plots are generated using an analytical method. The 'jacobian' function takes the string component entries from the entry boxes in the 'Field Input Frame' and calculates the Jacobian matrix using sympy partial differentiation. The resulting matrix is evaluated at the click coordinates (x_m, y_m) giving:

$$J = \begin{pmatrix} \frac{\partial u}{\partial x} & \frac{\partial u}{\partial y} \\ \frac{\partial v}{\partial x} & \frac{\partial v}{\partial y} \end{pmatrix}_{(x_m, y_m)} \tag{4}$$

For these plots, the 'dxg' and 'dyg' meshgrids must be centred on zero, rather than on the click coords used for the deriv and zoom plots (the first if statement in 'deriv_calc' accounts for this.) The component equations of the divergence and curl fields are then calculated:

$$u_{div} = \left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right)_{(x_m, y_m)} \Delta x \quad v_{div} = \left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right)_{(x_m, y_m)} \Delta y \tag{5}$$

$$u_{curl} = \left(\frac{\partial u}{\partial y} - \frac{\partial v}{\partial x}\right)_{(x_m, y_m)} \Delta y \quad v_{curl} = -\left(\frac{\partial u}{\partial y} - \frac{\partial v}{\partial x}\right)_{(x_m, y_m)} \Delta x \tag{6}$$

In this notation, Δx and Δy represent displacements within the inset axis. As a result, positive curl is shown by a linear field with counter-clockwise flow and negative by a clockwise flow, as would be given by the right hand rule. Additionally, the divergence and curl plots use scaling that matches the field, which ensures that the magnitude of the vectors varies appropriately.

0.5 Line Integrals

0.5.1 Lines

Upon selecting the 'line integrals' tab, with the polygons drop down selection, the user can draw lines on the plot by clicking to mark the successive start and end points as required. The users click coordinates are stored in a list (LI_coord). The function 'line_int_poly' takes the two most recent entries in the list of coordinates and finds the approximate line integral using the following method:

- Small displacements along the line in the x and y directions are calculated based on the last two entries in LI_coord (coords of points a and b) and the chosen number of intervals along the line, N. $dx = \frac{1}{N}(b_x - a_x)$, $dy = \frac{1}{N}(b_y - a_y)$
- A 2xN array 'intervals' is created and stores the coordinates of the points along the line which will be used in the final calculation. This is achieved using a for loop; $intervals(x_k) = a_x + kdx$, $intervals(y_k) = a_y + kdy$
- Next the vector field components must be evaluated at each of the interval points along the line. These components are stored in the 2xN array 'uv_store'.

- The line integral is approximated by summing the products of the small displacements dx and dy with the respective vector field components at each interval point. Since the displacements are the same along the straight line, the sum is simplified to:

$$\int_a^b \vec{F}\cdot\vec{dl} \approx \sum_{k=0}^{N-1}\vec{F}(x_k,y_k)\cdot\vec{dl}(x_k,y_k) = dx\sum_{k=0}^{N-1}F_x(x_k,y_k) + dy\sum_{k=0}^{N-1}F_y(x_k,y_k) \tag{7}$$

Here, $\vec{F} = F_x\ \hat{\imath} + F_y\ \hat{\jmath}$, $\vec{dl} = dx\ \hat{\imath} + dy\ \hat{\jmath}$ and (x_k, y_k) denotes the coordinates of the k^{th} interval point along the line. Each successive line integral is added to the total and displayed in a GUI label. The reset button clears the plot, the list of coordinates and the total, allowing the user to start from scratch.
This function operates not only over single starlight lines but also joins them head-to-tail to create polygonal shapes by accessing previous mouse click coordinates from LI_coord. The polygons can be closed by clicking near **any** previously clicked point. This is code by defining these coordinates as equal when the distance between them is smaller than a selected interval 'ctol'. Which is set to be 0.1 (in axis units).
This function also plots a square grid to aid the user in drawing rectangles on the plot, without the need for an additional function to draw squares. This grid is removed when any other option is selected.

0.5.2 Circles

Selecting the circles drop down option will allow the user to plot circles on the vector field, updating the position and radius (user selected by entry box) upon every click. Circles are centred on click location. The main difference from the calculation of straight line line integrals is that the displacement vectors are no longer constant, as they will be tangent to the circle at each interval point. The method is as follows:

- Define the small displacements dx and dy (arrays) for the N interval points using $dx_k = -Asin(kdt)$ and $dy_k = Acos(kdt)$. dt is a small angle increment and is stored as an array in the code (linspace from 0 to 2π with N points.) A, the magnitude of the displacement vector is chosen to be the length of the circumference swept out by each angle increment, $A = \frac{2\pi R}{N}$.
- Calculate the coordinates of interval points: $intervals(x_k) = x_m + Acos(kdt)$, $intervals(y_k) = y_m + Asin(kdt)$. (x_m, y_m) are the coordinates of the users click.
- Evaluate the vector components at each interval point as before.
- Find the total line integral using:

$$\oint \vec{F}\cdot\vec{dl} \approx \sum_{k=0}^{N-1}F_x(x_k,y_k)dx_k + \sum_{k=0}^{N-1}F_y(x_k,y_k)dy_k \tag{8}$$