Deadline Manager

IBP Group Project

Group Name	Student ID	Student Name	Group Leader
Little Genius	20216866	Minkun Yang	Minkun Yang
	20215417	Shang Wu	
	20215489	Jiayi Han	

CONTENTS

Introduction

Background

As Bani Ali et al. (2008) highlight, today's complex world requires people to have better time management skills. However, studies have shown that around 90% of students have different degrees of procrastination problems (Steel, 2007) and about 80% of employees have difficulties engaging in their work (Gallup, 2020). Consequently, there is a huge demand for personal productivity management tools: Fortune Business Insights (2019)'s report shows that the 2019's task management market is already valued at around $1.8 billion and it is projected to reach over $4.5 billion in 2026.

However, in current app stores, rarely can we find any software that is easy and interesting to use. Those apps seem to provide no ways for users to visualise the approaching of the deadlines and little or no incentive to keep improving their successful completion rates. It is easy for people to continuously add tasks onto a list, and then get lost in a long list. Therefore, we decide to develop our own time management program – Deadline Manager.

Deadline Manager

Its prime objective is to make time management easy and fun, and it achieves this mainly through 4 ways:

1. Prioritising all tasks using the focus matrix,
2. Visualising progress and deadlines through auto-updated progress bar and animation effect of the words,
3. Summarising all timelines using Gantt chart and,
4. Gamifying time management using cumulative history records and desktop assistant.

More detailed explanation of them will be provided in the second part. After that, we will discuss the program's potential impacts in the third section, before a conclusion is given at the end.

Programming

Login/Sign up

Code file: Deadline_Manager.py

Firstly, we design the (user interface) UI of the welcome screen (Figure 1), sign-up screen (Figure 2) and login screen (Figure 3) using Qt Designer and then use PyQt5.uic.loadUi() to load them.

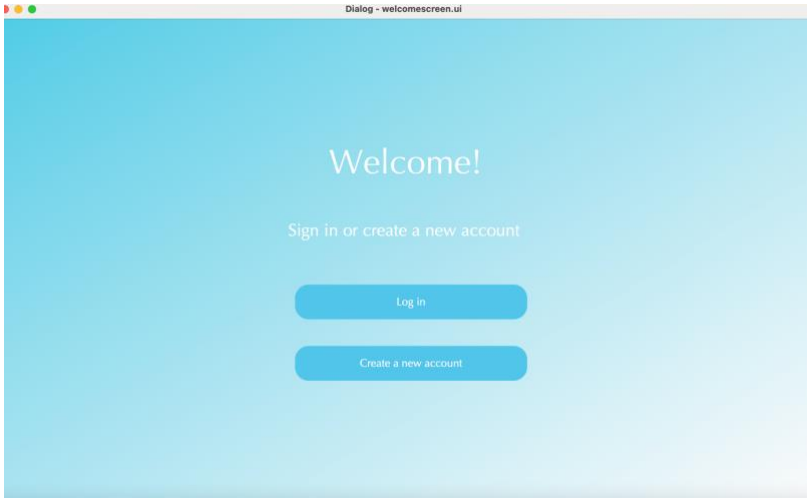


Figure 1 The welcome screen (welcomescreen.ui)

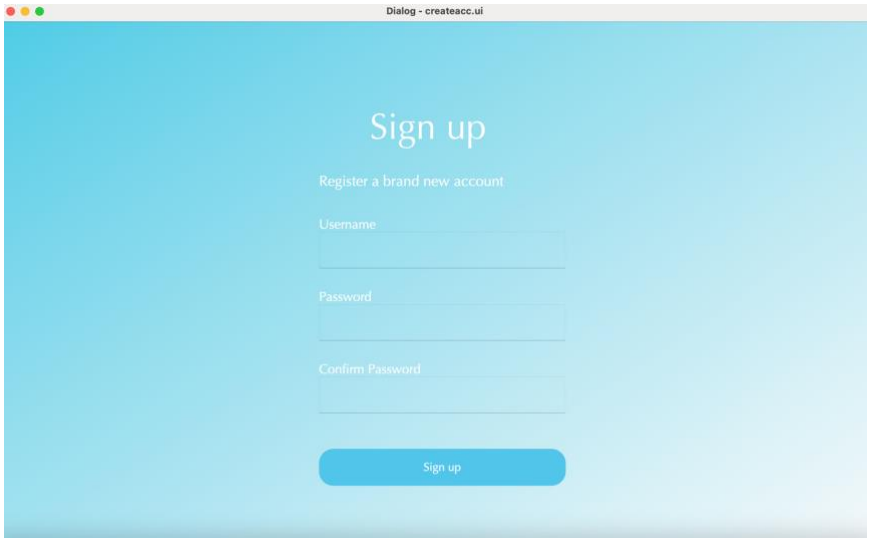


Figure 2 The sign up screen (createacc.ui)

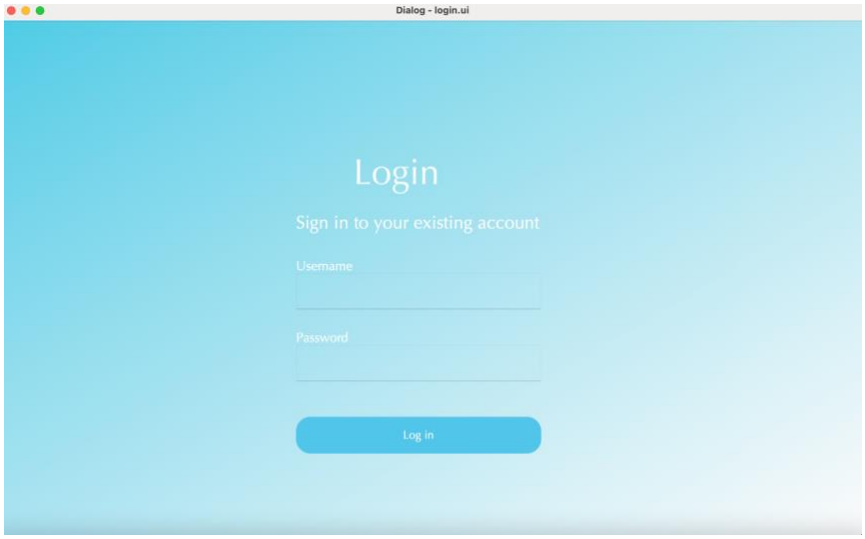


Figure 3 The log in screen (login.ui)

In our python file, we create three screen class for the 3 screens and

then link the buttons to their functions (Figure 4).

```
# create welcome screen class
class WelcomeScreen(QDialog):
    def __init__(self):
        super(WelcomeScreen, self).__init__()
        # load ui file
        loadUi("welcomescreen.ui", self)
        # connect login button to "direct to login" function
        self.login.clicked.connect(self.gotologin)
        # connect create a new account button to "direct to sign up" function
        self.create.clicked.connect(self.gotocreate)

    # direct to the login screen
    def gotologin(self):
        login = LoginScreen()
        widget.addWidget(login)
        widget.setCurrentIndex(widget.currentIndex() + 1)

    # direct to the sign up screen
    def gotocreate(self):
        create = CreateAccScreen()
        widget.addWidget(create)
        widget.setCurrentIndex(widget.currentIndex() + 1)

# create a login screen class
class LoginScreen(QDialog):
    def __init__(self):
        super(LoginScreen, self).__init__()
        loadUi("login.ui", self)
        # set password mode for the password input area
        self.passwordfield.setEchoMode(QtWidgets.QLineEdit.Password)
        self.login.clicked.connect(self.loginfunction)

    def loginfunction(self):
        user = self.emailfield.text()
        password = self.passwordfield.text()
        # judge whether the user has input all required fields
        if len(user) == 0 or len(password) == 0:
            self.error.setText("Please input all fields.")
```

Figure 4a Create class for welcome screen ad login screen

```
else:
    # connect to the user information database where there are all account and password records
    conn = sqlite3.connect("user_info.db")
    cur = conn.cursor()
    query = 'SELECT password FROM login_info WHERE username =\'' + user + "\'"

    cur.execute(query)
    result_pass = cur.fetchone()[0]
    # judge whether the password is correct
    if result_pass == password:
        PaintVar.userLogin = True
        # record user
        with open("./user.txt", "wb") as f:
            f.write(user.encode("utf-8"))
        # clear existing records on the main window
        main_window.clear()
        # add main window to the stacked widget
        widget.addWidget(main_window)
        # display users' existing to-do-list
        main_window.show_data()
        # draw a gantt chart according to the users' uncompleted tasks
        gantt()
        pixmap = QPixmap('./images/fig1.jpeg')
        # show gantt chart
        main_window.label_gantt_chart.setPixmap(pixmap)
        main_window.label_gantt_chart.setScaledContents(True)
        widget.setCurrentIndex(widget.currentIndex() + 1)

        '''设置自动登录'''
        global logged, login_action, task_action, matrix_action
        logged = True
        writeConfig('Auto login', 'username', str(user))
        login_action.setText("Sign Out[" + user + "]")
        task_action.setEnabled(True)
        matrix_action.setEnabled(True)

    else:
        self.error.setText("Invalid username or password")
```

Figure4b Create login class continued

```
# create a sign up screen class
class CreateAccScreen(QDialog):
    def __init__(self):
        super(CreateAccScreen, self).__init__()
        loadUi("createacc.ui", self)
        # set password mode for the password input area
        self.passwordfield.setEchoMode(QtWidgets.QLineEdit.Password)
        self.confirmpasswordfield.setEchoMode(QtWidgets.QLineEdit.Password)
        # link the button to the sign up function
        self.signup.clicked.connect(self.signupfunction)

    def signupfunction(self):
        user = self.emailfield.text()
        password = self.passwordfield.text()
        confirmpassword = self.confirmpasswordfield.text()
        # judge whether the user has input all required fields
        if len(user) == 0 or len(password) == 0 or len(confirmpassword) == 0:
            self.error.setText("Please fill in all inputs.")

        elif password != confirmpassword:
            self.error.setText("Passwords do not match.")
        else:
            # write new user's info into the database
            conn = sqlite3.connect("user_info.db")
            cur = conn.cursor()

            user_info = [user, password]
            cur.execute('INSERT INTO login_info (username, password) VALUES (?,?)', user_info)
            conn.commit()
            conn.close()
            login = LoginScreen()
            widget.addWidget(login)
            widget.setCurrentIndex(widget.currentIndex() + 1)
```

Figure 4c Create sign up screen class

We flip through the 3 screens using stacked widgets and we use sqlite3 to write and retrieve and user account and password information to and from the user_info.db file.

In the log in part (Figure 4b), if the user's account and password matches, the program will automatically direct to the main window.

Main Window

Code: main_window.py

Similarly, we designed its UI in Qt Designer (main_window_layout.ui) and convert it into a python file (main_window_layout.py).

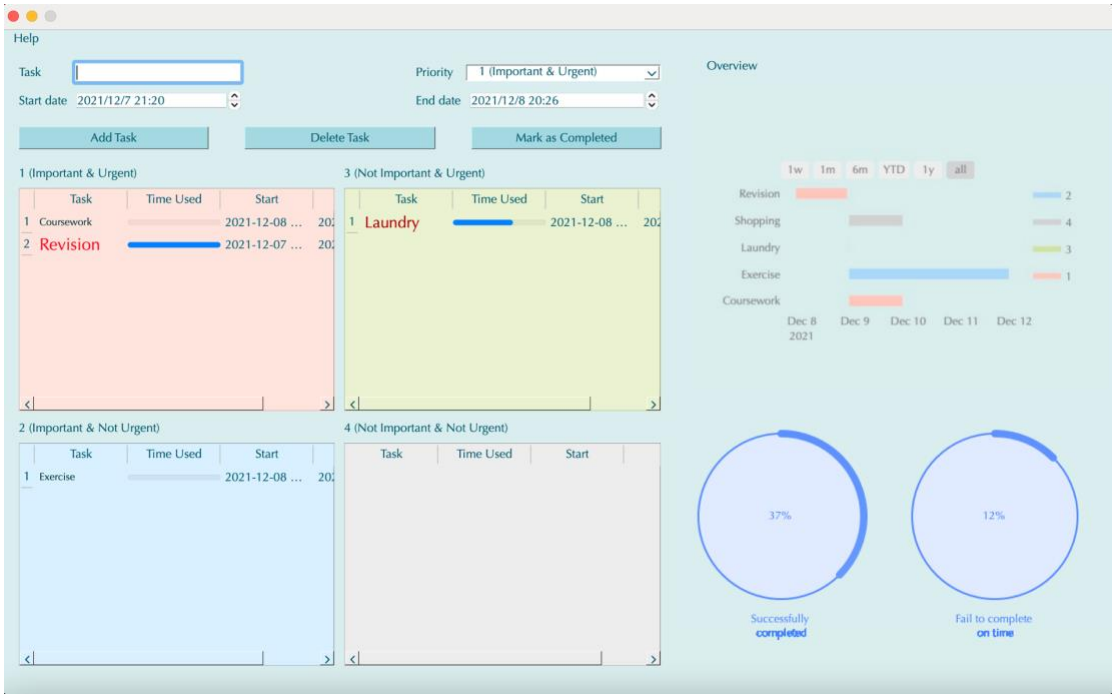


Figure 5 Main Screen

At the left top corner there is a Help menu bar (Figure 6), clicking on which, the users can see detailed information about how to use the program (Figure 7).

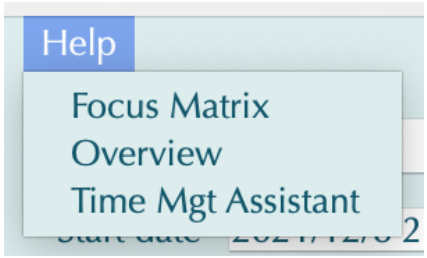


Figure 6 menu bar

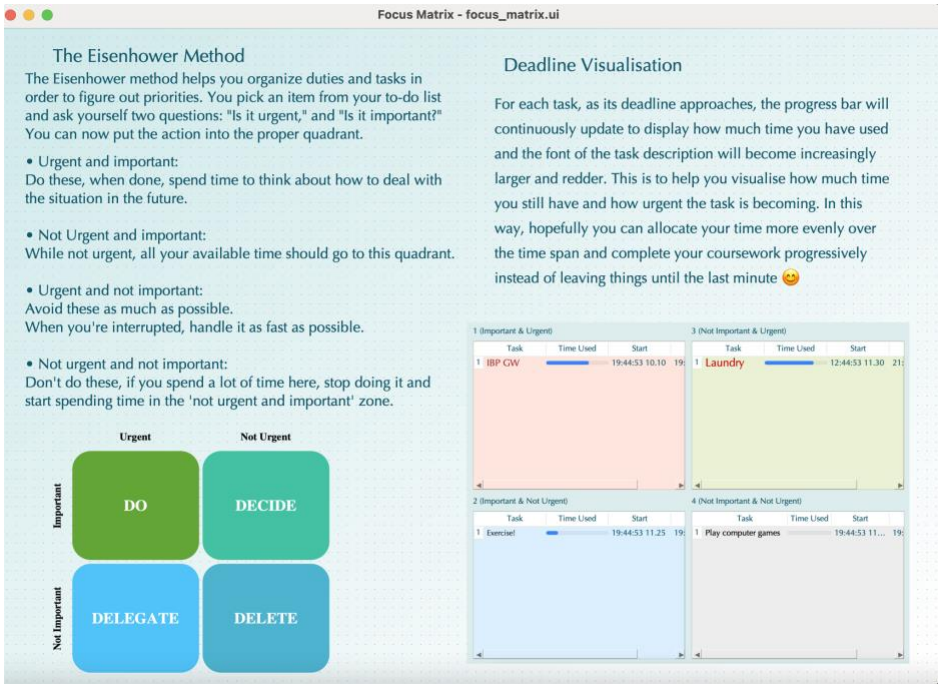


Figure 7a Detailed info about the focus matrix

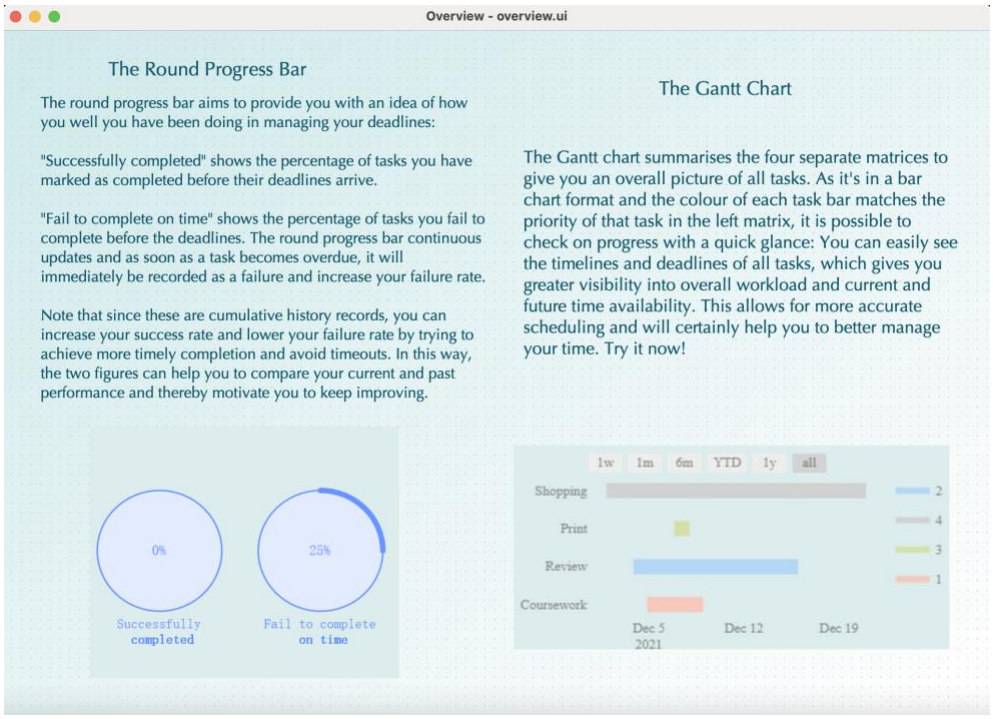


Figure 7b Detailed info about the overview section

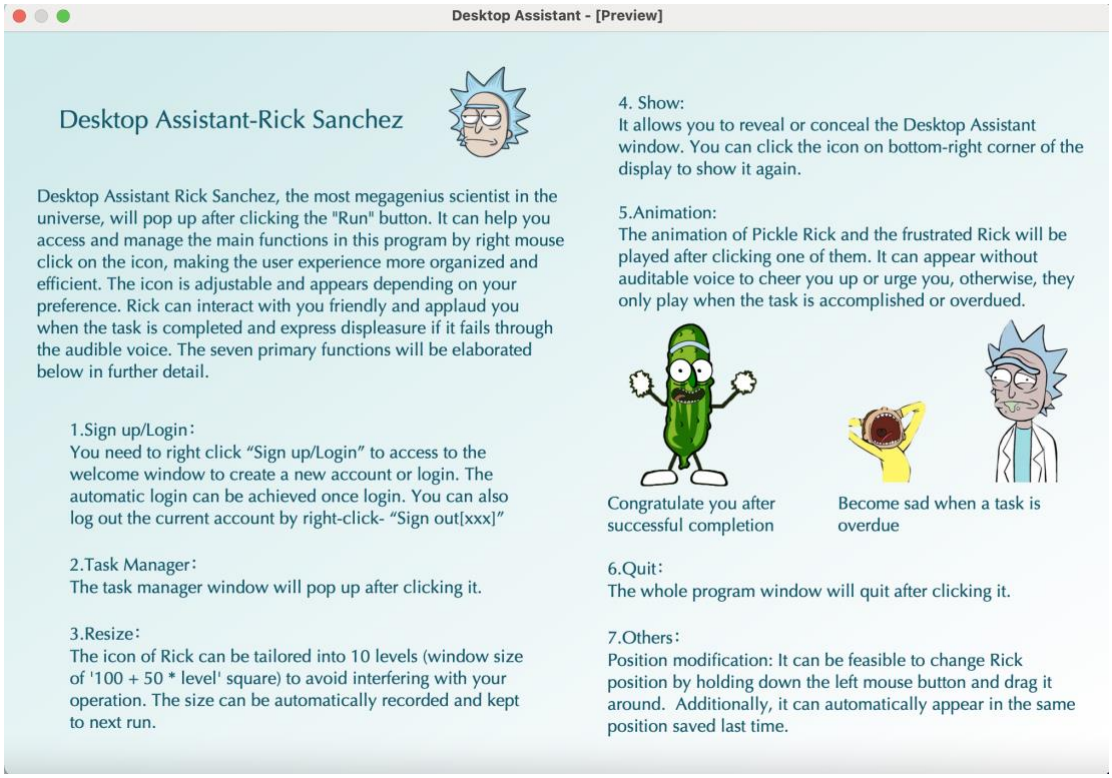


Figure 7c Detailed info about desktop assistant

```
# create a window class to display help information for the focus matrix
class FocusMatrix(QDialog):
    def __init__(self):
        super(FocusMatrix, self).__init__()
        loadUi("focus_matrix.ui", self)

# create a window class to display help information for the overview section
class Overview(QDialog):
    def __init__(self):
        super(Overview, self).__init__()
        loadUi("overview.ui", self)

# create a window class to display help information for the desktop assistant
class Assistant(QDialog):
    def __init__(self):
        super(Assistant, self).__init__()
        loadUi("assistant.ui", self)
```

Figure 8a Create help info screen class

```
# create the welcome screen and help info screens
main_window = MainWindow()
focus_matrix = FocusMatrix()
overview = Overview()
assistant = Assistant()
# link the actions under the help menu bar to their screens
main_window.actionFocus_Matrix.triggered.connect(focus_matrix.show)
main_window.actionOverview.triggered.connect(overview.show)
main_window.actionTime_Mgt_Assistant.triggered.connect(assistant.show)
```

Figure 8b Link the 3 help screens to the actions under the help menu bar

Focus matrix

According to Mfondoum *et al.* (2019), the focus matrix is a time management tool developed by Eisenhower in the 1950s, who mps all tasks along two dimensions -- importance and urgency. This helps to set priorities and employ different approaches to handle different types of tasks.

In our program, through the LineEdit widget and the QDateTimeEdit widget, the user can input the task name and time range. Meanwhile, he/she can also use the Combo Box on the right top corner to select the priority of the task (Figure 9). Then the program will record them into the data.db database and display them in the 4 Table widgets and Gantt chart correspondingly. This is achieved by linking the add button to the add function (Figure 10).

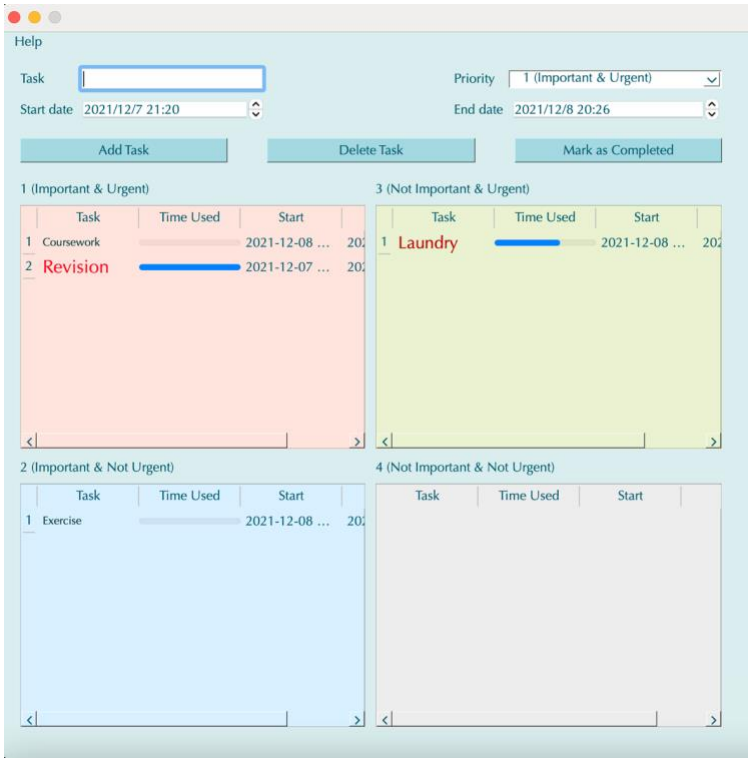


Figure 9 Focus matrix

```python
# create the main window class
class MainWindow(QMainWindow, Ui_MainWindow):
    def __init__(self, parent=None):
        super(MainWindow, self).__init__(parent)
        self.setupUi(self)
        # show the menu bar
        self.menuBar.setNativeMenuBar(False)
        # link 3 buttons to their functions
        self.pushButton_add.clicked.connect(self.add_task)
        self.pushButton_delete.clicked.connect(self.delete_task)
        self.pushButton_complete.clicked.connect(self.mark_task)
        # set the default time for dateTimeEdit widget
        self.dateTimeEdit_start.setDateTime(QtCore.QDateTime.currentDateTime())
        self.dateTimeEdit_end.setDateTime(QtCore.QDateTime.currentDateTime().addSecs(80))
```

Figure 10a connecting buttons to their functions

```
# define the add task function
def add_task(self):

    # get user entered item
    item_task = self.lineEdit_task.text()
    priority = self.comboBox_priority.currentText()
    start_date_q = self.dateTimeEdit_start.dateTime()
    end_date_q = self.dateTimeEdit_end.dateTime()

    # convert the time to string to store in the databsae
    item_start = start_date_q.toString('yyyy-MM-dd hh:mm:ss')
    item_end = end_date_q.toString('yyyy-MM-dd hh:mm:ss')

    # convert the QDateTime type in PyQt5 to integer
    # it is the number of seconds of the period from 1970-1-1 00:00:00 to the time point)
    pro_min = start_date_q.toTime_t()
    pro_max = end_date_q.toTime_t()
    # get current datetime
    now_q = QtCore.QDateTime.currentDateTime()
    now_int = now_q.toTime_t()

    # set up the progress bar
    time_used_visualisation = QtWidgets.QProgressBar(self.centralwidget)
    # the min value is the start time and the max value is the deadline
    time_used_visualisation.setRange(pro_min, pro_max)
    # let the progress bar shows how much has been used (from start time to current time)
    time_used_visualisation.setProperty("value", now_int)

    # add entered item in the table with the corresponding priority
    if priority == '1 (Important & Urgent)':
        row = self.tableWidget_1.rowCount()
        self.tableWidget_1.insertRow(row)
        item = QTableWidgetItem(item_task)
        self.tableWidget_1.setItem(row, 0, item)
        item = QTableWidgetItem(item_start)
        self.tableWidget_1.setItem(row, 2, item)
        item = QTableWidgetItem(item_end)
        self.tableWidget_1.setItem(row, 3, item)
        self.tableWidget_1.setCellWidget(row, 1, time_used_visualisation)
```

Figure 10b Add task function

```
    # add into the database
    item_class = 1
    dataset = Dataset()
    dataset.insert(item_class, item_task, item_start, item_end)
    dataset.close()


elif priority == "2 (Important & Not Urgent)":
    row = self.tableWidget_2.rowCount()
    self.tableWidget_2.insertRow(row)
    item = QTableWidgetItem(item_task)
    self.tableWidget_2.setItem(row, 0, item)
    item = QTableWidgetItem(item_start)
    self.tableWidget_2.setItem(row, 2, item)
    item = QTableWidgetItem(item_end)
    self.tableWidget_2.setItem(row, 3, item)
    self.tableWidget_2.setCellWidget(row, 1, time_used_visualisation)

    item_class = 2
    dataset = Dataset()
    dataset.insert(item_class, item_task, item_start, item_end)
    dataset.close()

elif priority == "3 (Not Important & Urgent)":
    row = self.tableWidget_3.rowCount()
    self.tableWidget_3.insertRow(row)
    item = QTableWidgetItem(item_task)
    self.tableWidget_3.setItem(row, 0, item)
    item = QTableWidgetItem(item_start)
    self.tableWidget_3.setItem(row, 2, item)
    item = QTableWidgetItem(item_end)
    self.tableWidget_3.setItem(row, 3, item)
    self.tableWidget_3.setCellWidget(row, 1, time_used_visualisation)

    item_class = 3
    dataset = Dataset()
    dataset.insert(item_class, item_task, item_start, item_end)
    dataset.close()
```

Figure 10c Add task function continued

After adding them, the user can use the delete button to remove the task from the table widget, Gantt chart and the database (Figure 11).

```
# define the delete task function
def delete_task(self):
    if self.tableWidget_1.selectedItems() != []:
        # get user deleted item
        task_info = self.tableWidget_1.selectedItems()[0].text()
        dataset = Dataset()
        # delete the record form the databasee
        dataset.delete(task_info)
        dataset.close()
    elif self.tableWidget_2.selectedItems() != []:
        task_info = self.tableWidget_2.selectedItems()[0].text()
        dataset = Dataset()
        dataset.delete(task_info)
        dataset.close()
    elif self.tableWidget_3.selectedItems() != []:
        task_info = self.tableWidget_3.selectedItems()[0].text()
        dataset = Dataset()
        dataset.delete(task_info)
        dataset.close()
    elif self.tableWidget_4.selectedItems() != []:
        task_info = self.tableWidget_4.selectedItems()[0].text()
        dataset = Dataset()
        dataset.delete(task_info)
        dataset.close()

    # remove the task from the to-do list table
    self.tableWidget_1.removeRow(self.tableWidget_1.currentRow())
    self.tableWidget_2.removeRow(self.tableWidget_2.currentRow())
    self.tableWidget_3.removeRow(self.tableWidget_3.currentRow())
    self.tableWidget_4.removeRow(self.tableWidget_4.currentRow())
```

Figure 11 Delete function

When the user completes the task, he/she can use the Mark as Completed button to update the status of the task in the database and remove it from the to-list table and Gantt chart (Figure 12).

```
# update the status to 1 (successful completion) when marking as completed if the task is not overdue
def mark_task(self):

    playsound('voice/rick-congrats.wav')

    if self.tableWidget_1.selectedItems() != []:
        # get the selected information
        task_info = self.tableWidget_1.selectedItems()[0].text()
        start_date = self.tableWidget_1.selectedItems()[1].text()
        end_date = self.tableWidget_1.selectedItems()[2].text()
        # update status in the database
        dataset = Dataset()
        dataset.update_status(task_info, start_date, end_date)
        dataset.close()
        # remove the task that has been completed from the to-do list
        self.tableWidget_1.removeRow(self.tableWidget_1.currentRow())

    # the same for the other 3 table widgets

    elif self.tableWidget_2.selectedItems() != []:
        task_info = self.tableWidget_2.selectedItems()[0].text()
        start_date = self.tableWidget_2.selectedItems()[1].text()
        end_date = self.tableWidget_2.selectedItems()[2].text()
        dataset = Dataset()
        dataset.update_status(task_info, start_date, end_date)
        dataset.close()
        self.tableWidget_2.removeRow(self.tableWidget_2.currentRow())
    elif self.tableWidget_3.selectedItems() != []:
        task_info = self.tableWidget_3.selectedItems()[0].text()
        start_date = self.tableWidget_3.selectedItems()[1].text()
        end_date = self.tableWidget_3.selectedItems()[2].text()
        dataset = Dataset()
        dataset.update_status(task_info, start_date, end_date)
        dataset.close()
```

Figure 12 Mark as Completed function

One differentiating function of the program is the animation effect of the task description. As the deadline approaches, the font of the task will become redder and larger. Combined with the continuously updated progress bar showing the percentage of time used, it can give the user a general feeling of how close the deadline is (i.e., to visualise the deadline) (Figure 13).

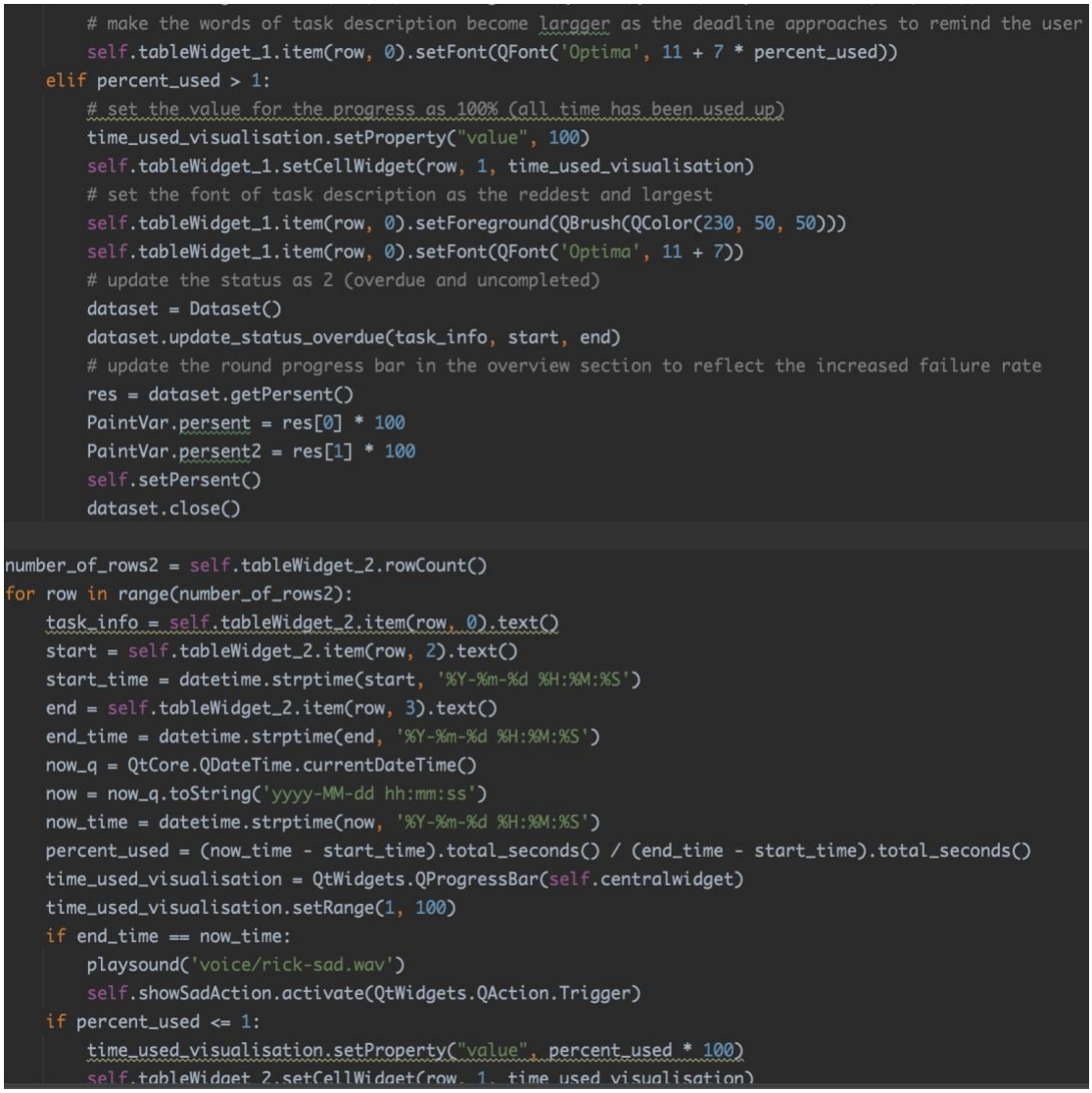
```
# define the update function
def deadline_is_coming(self):
    # update the gantt chart
    gantt()
    pixmap = QPixmap('./images/fig1.jpeg')
    self.label_gantt_chart.setPixmap(pixmap)
    self.label_gantt_chart.setScaledContents(True)
    # update each task record
    number_of_rows1 = self.tableWidget_1.rowCount()
    for row in range(number_of_rows1):
        # get the existing information
        task_info =self.tableWidget_1.item(row, 0).text()
        start = self.tableWidget_1.item(row, 2).text()
        # convert string type to datetime type
        start_time = datetime.strptime(start, '%Y-%m-%d %H:%M:%S')
        end = self.tableWidget_1.item(row, 3).text()
        end_time = datetime.strptime(end, '%Y-%m-%d %H:%M:%S')
        now_q = QtCore.QDateTime.currentDateTime()
        now = now_q.toString('yyyy-MM-dd hh:mm:ss')
        now_time = datetime.strptime(now, '%Y-%m-%d %H:%M:%S')

        # calculate the percentage of time used
        percent_used = (now_time - start_time).total_seconds() / (end_time - start_time).total_seconds()

        # create a progress bar with range form 1 to 100
        time_used_visualisation = QtWidgets.QProgressBar(self.centralwidget)
        time_used_visualisation.setRange(1, 100)

        if end_time == now_time:
            playsound('voice/rick-sad.wav')
            self.showSadAction.activate(QtWidgets.QAction.Trigger)
        # judge whether the task is overdue
        if percent_used <= 1:
            # update the progress bar
            time_used_visualisation.setProperty("value", percent_used*100)
            self.tableWidget_1.setCellWidget(row, 1, time_used_visualisation)
            # make the words of task description become redder as the deadline approaches too remind the user
```

Figure 13a Update function



```
        # make the words of task description become largger as the deadline approaches to remind the user
        self.tableWidget_1.item(row, 0).setFont(QFont('Optima', 11 + 7 * percent_used))
    elif percent_used > 1:
        # set the value for the progress as 100% (all time has been used up)
        time_used_visualisation.setProperty("value", 100)
        self.tableWidget_1.setCellWidget(row, 1, time_used_visualisation)
        # set the font of task description as the reddest and largest
        self.tableWidget_1.item(row, 0).setForeground(QBrush(QColor(230, 50, 50)))
        self.tableWidget_1.item(row, 0).setFont(QFont('Optima', 11 + 7))
        # update the status as 2 (overdue and uncompleted)
        dataset = Dataset()
        dataset.update_status_overdue(task_info, start, end)
        # update the round progress bar in the overview section to reflect the increased failure rate
        res = dataset.getPersent()
        PaintVar.persent = res[0] * 100
        PaintVar.persent2 = res[1] * 100
        self.setPersent()
        dataset.close()

number_of_rows2 = self.tableWidget_2.rowCount()
for row in range(number_of_rows2):
    task_info = self.tableWidget_2.item(row, 0).text()
    start = self.tableWidget_2.item(row, 2).text()
    start_time = datetime.strptime(start, '%Y-%m-%d %H:%M:%S')
    end = self.tableWidget_2.item(row, 3).text()
    end_time = datetime.strptime(end, '%Y-%m-%d %H:%M:%S')
    now_q = QtCore.QDateTime.currentDateTime()
    now = now_q.toString('yyyy-MM-dd hh:mm:ss')
    now_time = datetime.strptime(now, '%Y-%m-%d %H:%M:%S')
    percent_used = (now_time - start_time).total_seconds() / (end_time - start_time).total_seconds()
    time_used_visualisation = QtWidgets.QProgressBar(self.centralwidget)
    time_used_visualisation.setRange(1, 100)
    if end_time == now_time:
        playsound('voice/rick-sad.wav')
        self.showSadAction.activate(QtWidgets.QAction.Trigger)
    if percent_used <= 1:
        time_used_visualisation.setProperty("value", percent_used * 100)
        self.tableWidget_2.setCellWidget(row, 1, time_used_visualisation)
```

Figure 13b Update function continued

Overview

Gantt Chart

According to Sharon and Dori (2017), Gantt chart is a bar chart originally designed by Henry Gantt for project scheduling. In our program, we adapt it to summarise the four separate matrices to give the user an overall picture (Figure 14). Since it displays all tasks along the horizontally timeline in a bar chart format, it is possible for the user to check on progress with a quick glance: He/she can easily track all the timelines and deadlines, which gives him/her greater visibility into overall workload and time availability. Presumably, this could allow for more accurate scheduling.

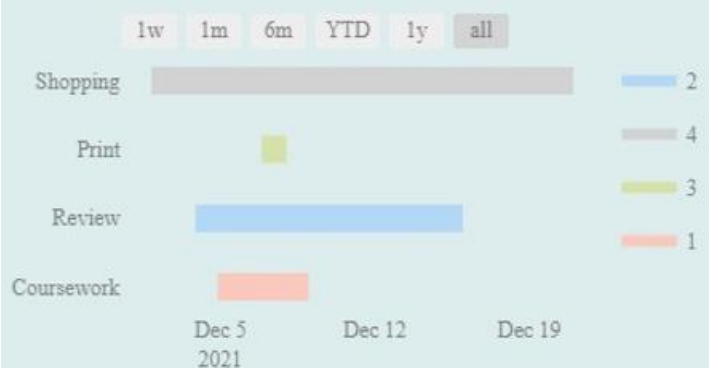


Figure 14 Gantt chat example

As the chart.py file shows, we realise this function through drawing the Gantt chart using the Plotly library and then save it as a jpeg image named 'gantt.jpeg' under the images directory (Figure 15). In the main_window.py, we continuously update it through calling it in the update function (deadline_is_coming) and display it using label.setPixmap() method (Figure 16).

```
from db_controller import *
import plotly.express as px
import plotly.figure_factory as ff
import pandas as pd
import os

# we use plotly to draw the gantt chart
# reference https://plotly.com/python/gantt/

def gantt():
    # specify the colours for 4 types of priority
    colors = {'1': 'rgb(250, 200, 189)',
              '2': 'rgb(179, 215, 247)',
              '3': 'rgb(213, 225, 168)',
              '4': 'rgb(210, 210, 211)'}

    # create a directory to store the gantt chart
    if not os.path.exists("images"):
        os.mkdir("images")
    # get the current logged in user
    user = ""
    with open("./user.txt", "r") as f:
        user = f.read()
    if user == '':
        print('please login first')
    else:
        # get the users' uncompleted tasks from the database
        con = sqlite3.connect('./data/database.db')
        df = pd.read_sql_query("SELECT * from TaskTable  WHERE (status = 0 OR status = 2) AND user = '%s'" % user, con)
        # judge whether the user has existing uncompleted tasks
        if df.empty:
            # draw an empty chart if there is no existing uncompleted tasks
            fig = px.timeline(df, x_start="start_date", x_end="end_date", y="task_info", labels={"task_info": None},)
            fig.update_yaxes(autorange="reversed")
            # customise the layout of the  chart
            fig.update_layout(paper_bgcolor='rgb(221, 237, 238)')
            fig.update_layout(plot_bgcolor='rgb(221, 237, 238)')
            # store the chart as a jpeg image
            fig.write_image("images/fig1.jpeg")
        else:
```

Figure 15a Draw the Gantt chart

```
    fig.write_image("images/fig1.jpeg")
else:
    # draw a gantt chart for existing uncompleted tasks
    # convert the sqlite3 data into a dataframe
    df['task_class'] = df['task_class'].astype("string")
    df2 = pd.DataFrame()
    df2['Task'] = df['task_info']
    df2['Start'] = df['start_date']
    df2['Finish'] = df['end_date']
    df2['Resource'] = df['task_class']

    # draw gantt chart
    # colour index is the number for 4 types of priority
    fig = ff.create_gantt(df2, colors=colors, index_col='Resource', show_colorbar=True,
                          group_tasks=True, showgrid_x=False, showgrid_y=False, title=None)
    # customise the layout of the  chart
    fig.update_yaxes(autorange="reversed")
    fig.update_layout(paper_bgcolor='rgb(221, 237, 238)')
    fig.update_layout(plot_bgcolor='rgb(221, 237, 238)')
    fig.update_layout(
        font_family="Optima",
        font_color="rgb(127, 127, 127)",
    )
    fig.update_layout(
        autosize=False,
        width=421,
        height=321)
    # store the chart as a jpeg image
    fig.write_image("images/gantt.jpeg")
```

Figure 15b Draw the Gantt chart continued

```
# define the update function
def deadline_is_coming(self):
    # update the gantt chart
    gantt()
    pixmap = QPixmap('./images/fig1.jpeg')
    self.label_gantt_chart.setPixmap(pixmap)
    self.label_gantt_chart.setScaledContents(True)
```

Figure 16 Update and display the Gantt chart

Round Progress Bar

The other part of the overview consists of two round progress bars: "Successfully completed" shows the percentage of tasks marked as completed before the deadlines;

"Fail to complete on time" shows the percentage of tasks completed out of time (Figure 17).

The round progress bar continuously updates and once a task becomes overdue, the failure rate will increase to reflect that. Since these are cumulative history records, the users could be motivated to achieve more timely completion and avoid timeouts. In this way, the two figures can help users to compare their current and past performance and thereby motivate them to keep improving.

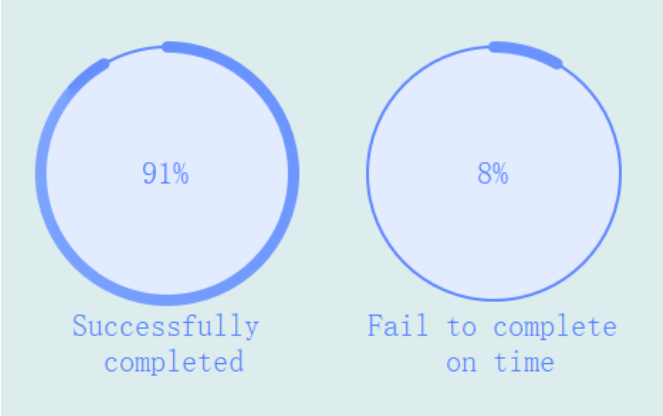


Figure 17 Round Progress Bar example

We achieve this by counting the numbers in the database and drawing them on a widget. Since the database has already updated task's status automatically in the previous steps, we can easily get the three required numbers through count statement of SQLite (Figure 18). Then we display the round progress bar with the set parameters by defining a paint function using canvas in the main_window.py and continuously update it (Figure 19).

```
def getPercent(self):
    # get the name of the currently logged in user
    user = ""
    with open("./user.txt", "r") as f:
        user = f.read()
    # get quantity of total tasks
    sql="select count(*) from tasktable where user=?"
    # get quantity of tasks completed on time
    sql2="select count(*) from tasktable where status=1 and user=?"
    # get quantity of tasks overdue
    sql3="select count(*) from tasktable where (status=2 or status=3) and user=?"
    # connect to the database and execute the sql
    cur = self.conn.cursor()
    cur2 = self.conn.cursor()
    cur3 = self.conn.cursor()
    percent1=0
    percent2=0
    try:
        data1 = cur.execute(sql, (user,))
        data2 = cur2.execute(sql2, (user,))
        data3 = cur3.execute(sql3, (user,))
        # deal with situations where the total number of tasks is zero
        if (data1!=None):
            all_num=data1.fetchone()[0]
            if(data2!=None):
                status1_num=data2.fetchone()[0]
                percent1=(status1_num/all_num)
            if(data3!=None):
                percent2=(data3.fetchone()[0]/all_num)
```

Figure 18a Get required data from database

```
# initialize canvas for round progress bar
self.percent = PaintVar.percent
self.percent2 = PaintVar.percent2
canvas = QtGui.QPixmap(480, 281)
# set the background color of round progress bar to the same colour as the main window
canvas.fill(QColor("#DDEDED"))
self.label_2.setPixmap(canvas)

# connect to database to get percent data
dataset = Dataset()
res = dataset.getPercent()
PaintVar.percent = res[0] * 100
PaintVar.percent2 = res[1] * 100
# execute setPercent function to update present percentage
self.setPercent()
dataset.close()
```

Figure 18b Update present percentage data

```
def paintEvent(self, event):
    rotateAngle = 360 * self.percent / 100
    rotateAngle2 = 360 * self.percent2 / 100
    painter = QtGui.QPainter(self.label_2.pixmap())
    painter.eraseRect(0, 0, 480, 281)
    painter.setRenderHints(QtGui.QPainter.Antialiasing)
    # customize the brush parameters
    painter.setPen(QtCore.Qt.NoPen)
    painter.setBrush(QBrush(QColor("#DDEDEE")))
    painter.drawRect(0, 0, 480, 281)
    painter.setBrush(QBrush(QColor("#5F89FF")))
    painter.drawEllipse(PaintVar.left, PaintVar.top, PaintVar.progressR,
                        PaintVar.progressR)
    # Draw the outer circle
    painter.drawEllipse(PaintVar.left + 230, PaintVar.top, PaintVar.progressR, PaintVar.progressR)
    painter.setBrush(QBrush(QColor("#e3ebff")))
    painter.drawEllipse(PaintVar.left + 2, PaintVar.top + 2, PaintVar.progressR - 4,
                        PaintVar.progressR - 4)
    # Draw inner circle
    painter.drawEllipse(PaintVar.left + 232, PaintVar.top + 2, PaintVar.progressR - 4, PaintVar.progressR - 4)
    gradient = QConicalGradient(50, 50, 91)
    gradient.setColorAt(0, QColor("#95BBFF"))
    gradient.setColorAt(1, QColor("#5C86FF"))
```

Figure 19a Draw the round progress bar (basic shape)

```
# plot the progresses according to the percentage of completed and overdue respectively
painter.drawArc(QtCore.QRectF(PaintVar.left + 1, PaintVar.top + 1, PaintVar.progressR - 2, PaintVar.progressR - 2),(90 - 0) * 16, -rotateAngle * 16)
painter.drawArc(QtCore.QRectF(PaintVar.left + 231, PaintVar.top + 1, PaintVar.progressR - 2, PaintVar.progressR - 2),(90 - 0) * 16, -rotateAngle2 * 16)
font = QtGui.QFont()
font.setFamily("Optima")
font.setPointSize(13)
painter.setFont(font)
painter.setPen(QColor("#5481FF"))
painter.drawText(
    QtCore.QRectF(PaintVar.left + 1, PaintVar.top + 1, PaintVar.progressR - 4, PaintVar.progressR - 4),
    Qt.AlignCenter,
    "%d%%" % self.percent)
# Show current progress
painter.drawText(
    QtCore.QRectF(PaintVar.left + 1, PaintVar.top + 120, PaintVar.progressR - 4, PaintVar.progressR - 4),
    Qt.AlignCenter,
    "Successfully \n completed")
# progress name
painter.drawText(
    QtCore.QRectF(PaintVar.left + 231, PaintVar.top + 1, PaintVar.progressR - 4, PaintVar.progressR - 4),
    Qt.AlignCenter,
    "%d%%" % self.percent2)
# Show current progress
painter.drawText(
    QtCore.QRectF(PaintVar.left + 231, PaintVar.top + 120, PaintVar.progressR - 4, PaintVar.progressR - 4),
    Qt.AlignCenter,
    "Fail to complete \n on time")
```

Figure 19b Draw the round progress bar (dynamic progress)

```
# set the brush parameter
# it will only be called when the user has logged in successfully
class PaintVar:
    userLogin=False
    percent=0
    percent2=0
    winw=300
    winh=0
    top=20
    left=30
    progressR=180
```

Figure 19c Make sure that the progress bar will only execute when logging in successfully

Desktop Assistant

Window drawing

The window is drawn by PyQt5. The mask is set for the window to

make the background transparent and draw the graphical window using QLabel. Meanwhile, the window is set without a frame and always on the top.

Window movement

Due to the cancellation of the window frame, it is not available to drag the title bar to move the window. Hence, the window movement method needs to be rewritten. It can be achieved by using the mouse events: when the left mouse button is pressed and then moved, the window is moved to the mouse position accordingly.

```
"""Window move"""
def mousePressEvent(self, event):
    if event.button() == Qt.LeftButton:
        self.is_follow_mouse = True
        self.mouse_drag_pos = event.globalPos() - self.pos()
        event.accept()

def mouseMoveEvent(self, QMouseEvent):
    if Qt.LeftButton and self.is_follow_mouse:
        self.move(QMouseEvent.globalPos() - self.mouse_drag_pos)
        QMouseEvent.accept()
```

Resize

The QInputDialog.getItem is leveraged to input the size level of the window (set in the range 1-10). According to the size level selected, the window's mask and the graphic image will be scaled and the window gets repainted.

Show window

It is to set whether the window is visible. When the desktop window is hidden, you can right-click the icon in the tray to display it.

System Tray

```
"""Add system tray and events"""
def trayEvent(self):
    tray = QSystemTrayIcon(self)
    tray.setIcon(QIcon('images/Rick-Icon.png'))
    tray_menu = QMenu()

    tray_menu.addAction(visible_action)

    '''Quit'''
    quit_action = QAction("Quit", self)
    quit_action.triggered.connect(self.quit)
    tray_menu.addAction(quit_action)

    tray.setContextMenu(tray_menu)
    tray.show()
```

QSystemTrayIcon is leveraged to set the tray icon for the desktop assistant, and right-click menu is set to it to realize the visibility and "Quit".

Read and write configuration

The configparser library is to read and write 'ini' files and the relevant parameters will be written into the configuration file. When the program is run next time, the configuration will be automatically read to realize automatic login and keep the preference of window.

```
def loadConfig():
    global sizeX, window_x, window_y, loggedUser, logged
    config = configparser.ConfigParser()
    config.read('config.ini')
    sizeX = int(config.get('config', 'window_size'))
    if config.get('config', 'window_x') == 'null' or config.get('config', 'window_y') == 'null':
        window_x = None
        window_y = None
    else:
        window_x = int(config.get('config', 'window_x'))
        window_y = int(config.get('config', 'window_y'))

    if config.get('Auto login', 'username') == "null":
        loggedUser = None
    else:
        loggedUser = str(config.get('Auto login', 'username'))
        logged = True

def writeConfig(section, option, key):
    config = configparser.ConfigParser()
    config.read('config.ini')
    config.set(section, option, key)
    config.write(open('config.ini', "r+"))
```

Interactive animation

There are two interactions of the desktop assistant:

1. when users mark the task as completed before the deadline, it will play the congrats interaction;
2. When the task becomes overdue, it will play the sad interaction.

The animation is implemented by setting a 100-millisecond timer and calling the updated function in every timeout situation. The QLabel is leveraged to load the animation sequence frame.

```
"""Show Sad Rick"""
@pyqtSlot()
def showSad(self):
    global sizeX, sads, sadNow, sadCycle, resize_action, sad_animation

    sadNow = 0
    sadCycle = 0

    resize_action.setEnabled(False)
    sad_animation.setEnabled(False)
    pickle_animation.setEnabled(False)
    self.timer.timeout.connect(self.updateSad)

    self.mask = QBitmap.fromImage(
        QImage("images/sad/Full-Mask.png").scaled(QSize(120 * int(sizeX), 90 * int(sizeX))))
    self.setMask(self.mask)
    self.resize(self.mask.size())
    self.label.setPixmap(sads[0])
    self.label.resize(QSize(120 * int(sizeX), 90 * int(sizeX)))

    self.setAutoFillBackground(False)
    self.setAttribute(Qt.WA_TranslucentBackground, True)
    self.timer.start(100)
    sad_animation.triggered.disconnect()
    sad_animation.triggered.connect(self.showSad)
```

```
@pyqtSlot()
def updateSad(self):

    global sads, sadNow, sadCycle, resize_action, animation_action, pickle_animation

    self.label.setPixmap(sads[sadNow])

    if sadCycle == 1:

        self.timer.timeout.disconnect()
        self.showRick01()
        resize_action.setEnabled(True)
        pickle_animation.setEnabled(True)
        sad_animation.setEnabled(True)
    else:
        if sadNow < len(sads) - 1:
            sadNow += 1
        else:
            sadNow = 0
            sadCycle += 1
```

Play voice
This function leverages the pygame library.

```
def playsound(path):
    pygame.mixer.init()
    sound = pygame.mixer.Sound(path)
    sound.play(0)
```

Impacts

Business:
As cited in the introduction, there is a probably underserved time management software market that worth billions of dollars. With progress visualisation and time management gamification being its most distinguishing features, our Deadline Manager has the competitive advantage of being simple and fun-to-use and it possibly has the potential to fill this market gap.

Social:
Besides the prevalence of poor time management problem, research has also shown how procrastination has affected our lives adversely: Data from Steel (2007) reveals that over 94% respondents experience anxiety, frustration, depression and anger due to procrastination. Furthermore, a survey conducted by Jones and Blankenship (2021) underlines that students with procrastination problems performed significantly worse than their counterparts.

Through Making management less complicated and painful, our program seeks to improve people's life quality by helping them overcome procrastination and be freed from its negative impacts on emotion, health, studies and work.

Word count: 1480 words (exclude annotations)

References

Bani Ali, A. S., Anbari, F. T. and Money, W. H. (2008) 'Impact of Organizational and Project Factors on Acceptance and Usage of Project Management Software and Perceived Project Success', *Project Management Journal*, 39, pp. 5-33.

Fortune Business Insights (2019) *Task Management Software Market Size, Share & Industry Analysis and Regional Forecast, 2019-2026.* Available at: https://www.fortunebusinessinsights.com/task-management-software-market-102249 (Accessed: 13 December 2021).

Gallup (2020) *State of the Global Workplace 2021 Report.* Available at: https://www.gallup.com/workplace/349484/state-of-the-global-workplace.aspx (Accessed: 13 December 2021).

Jones, I. and Blankenship, D. (2021) 'Year two: Effect of procrastination on academic performance of undergraduate online students', *Research in Higher Education Journal*, 39.

Mfondoum, A. N., Tchindjang, M., Mfondoum, J. M., and Makouet, I. (2019) 'Eisenhower matrix* Saaty AHP= Strong actions prioritization? Theoretical literature and lessons drawn from empirical evidences' *IAETSD-Journal for Advanced Research in Applied Sciences*, 6(2), pp. 13-27.

Sharon, A. and Dori, D. (2017) 'Model-Based Project-Product Lifecycle Management and Gantt Chart Models: A Comparative Study'. *Systems Engineering*, 20, pp. 447-466.

Steel, P. (2007) 'The nature of procrastination: a meta-analytic and theoretical review of quintessential self-regulatory failure', *Psychological bulletin,* 133, pp. 65-94.