TDT4127 Programming and Numerics
Week 46/47

Repetition and exam preparation
Next week

• Questions about the exam:
  – Friday November 23, 16:15-17:00
  – Bring your questions, Guttorm and I will bring our answers
  – Afterward, 17:00 – 18:00: Final exam prep
Today

• Finalize adaptive Simpson’s method
  – Going through implementation
• Repetition
  – Summarize what we’ve learned
  – Go through auditorium exercise 2
• Exam preparation

• **Question:** 15 minute break at 17:00?
Implementing Adaptive Simpson’s rule

\( S(a, b) \) denotes Simpson’s on the integral from \( a \) to \( b \). To approximate the integral over \([a, b]\) with error < \( \epsilon \):

1. Compute \( S(a, b) \).
2. Compute \( S(a, c) \) and \( S(b, c) \).
3. Estimate the error in \( S(a, c) + S(b, c) \):
   
   \[
   \text{if } |S(a, b) - (S(a, c) + S(b, c))| < 15 \cdot \epsilon:
   \]
   
   \[
   \text{return } \frac{16}{15} (S(a, c) + S(b, c)) - \frac{1}{15} S(a, b)
   \]

   else:

   estimate the integrals over \([a, c]\) and \([c, b]\) with error less than \( \epsilon/2 \)
   
   return the two estimates added together
Repetition
Week 35/36: Number representation

- Computers mainly use two storage formats for numbers: Integers and floating point numbers (floats)
- **Integers**: *Precise* representations of whole numbers
  - Used for *counting*, *numbering* etc.
  - **Format**: Binary numbers. 8-bit example:
    
    \[
    10010101 = 1\times128 + 0\times64 + 0\times32 + 1\times16 + 0\times8 + 1\times4 + 0\times2 + 1\times1 = 149
    \]
  - More bits ⇔ can represent larger numbers
  - First bit may represent the sign (0 means negative, 1 positive)
Week 35/36: Number representation

• **Floating point numbers**: *Imprecise* versions of real numbers
  – Used in *calculations* requiring *decimal points*
  – **Format**: Scientific notation in base 2 (total system)
    
    \[a = (-1)^{sg} \times 2^{e-b} \times 1.s_1s_2s_3 ... s_K\]

    • *sg*: sign,  *e*: exponent,  *b*: bias,  \[1.s_1s_2s_3 ... s_K\]: significand/mantissa
  – Due to imprecision, be careful with floating point operations:
    • *a \pm b* is problematic if *a* and *b* are very different in size
    • *a \times b* and *a/b* are safe
    • *a == b* is very unsafe and should be avoided (check \(|a - b| < \varepsilon\) instead)
Week 36/38/39: Equation solvers

- Solving \( f(x) = g(x) \) ⇔ solving \( h(x) = f(x) - g(x) = 0 \)
  - Therefore the algorithms are based on solving \( h(x) = 0 \).

- Three methods: bisection, secant and Newton’s
  - Newton uses derivative. Secant and bisection: derivative free
  - Newton is faster than secant which is faster than bisection
  - Bisection has less rigid restrictions than secant which has less rigid restrictions than Newton

<table>
<thead>
<tr>
<th>Property type</th>
<th>Newton’s method</th>
<th>Secant method</th>
<th>Bisection method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Continuity</td>
<td>( f'' )</td>
<td>( f' )</td>
<td>( f )</td>
</tr>
<tr>
<td>Nonzero</td>
<td>( f''(z) \neq 0, f'(x) \neq 0 )</td>
<td>( f'(z) \neq 0 )</td>
<td>None</td>
</tr>
<tr>
<td>Extra bounds</td>
<td>( \left</td>
<td>\frac{f''(x)}{f'(y)} \right</td>
<td>\leq A )</td>
</tr>
<tr>
<td>Starting point</td>
<td>Close enough</td>
<td>Close enough</td>
<td>([a, b]) encloses ( z )</td>
</tr>
</tbody>
</table>
Algorithm: **Bisection method**

- **Type:** Equation solver. Finds zeroes: \( f(x) = 0 \)
- **Initialization:** \([a, b]\) such that \( f(a) \) and \( f(b) \) have different signs \( (f(a)f(b) < 0) \), a minimum width \( \epsilon \).
- **Mathematically:** Halve the interval, but ensure \( f(a)f(b) < 0 \)
- **Pseudoalgorithm:**

  ```python
  while abs(a-b) > epsilon:
      c = (a+b)/2
      if f(a) and f(c) have the same sign:
          a = c
      else:
          b = c
      if f(c) is 0:
          return c
  return c
  ```
Algorithm: **Newton’s method**

• **Type:** Equation solver. Finds zeroes: \( f(x) = 0 \)

• **Initialization:** Starting value \( x_0 \), tolerances \( \epsilon, \delta \).

• **Mathematically:** \( x_{k+1} = x_k - f(x_k)/f'(x_k) \)

• **Algorithm:**

  \[
  \begin{align*}
  k &= 0 \\
  \text{diff} &= \text{delta} + 1 \\
  \text{while } f(x_k) > \text{epsilon} \text{ and } \text{diff} > \text{delta} & \Rightarrow \\
  x_{k+1} &= x_k - f(x_k)/f'(x_k) \\
  \text{diff} &= x_{k+1} - x_k \\
  k &= k + 1 \\
  \text{return } x_{k+1}
  \end{align*}
  \]

• **Note:** Requires the derivative \( f'(x) \)
Algorithm: **Secant method**

- **Type:** Equation solver. Finds zeroes: \( f(x) = 0 \)
- **Initialization:** Starting values \( x_0 \) and \( x_1 \), tolerances \( \epsilon, \delta \).
- **Mathematically:**
  \[
  x_{k+1} = x_k - f(x_k) \frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})}
  \]
- **Algorithm:**
  
  \[
  k = 1
  \]
  
  \[
  \text{while } f(x_k) > \text{epsilon and abs}(x_k - x_{k-1}) > \delta \]
  
  \[
  x_{k+1} = x_k - f(x_k)(x_k - x_{k-1})/(f(x_k) - f(x_{k-1}))
  \]
  
  \[
  k = k+1
  \]
  
  \[
  \text{return } x_{k+1}
  \]
- **Note:** Can be seen as a derivative-free version of Newton’s
Week 37/45: Numerical integration

- **Task:** Compute a definite integral \( \int_{a}^{b} f(x) \, dx \)
- **Three methods:** Midpoint, trapezoidal, Simpson’s rule
  - Based on: constant, linear and quadratic approximations of \( f \).
  - Simpson’s rule is a bit more work but also more accurate
- **Composite** methods: Split \([a, b]\) into \( N \) parts, integrate each part separately, add together.
- **Error analysis**, \( M_2 = \max_{a \leq y \leq b} f''(y), M_4 = \max_{a \leq y \leq b} f''''(y) \):
  \[ E_{MP} \leq \frac{(b - a)^3}{24N^2} M_2, \quad E_{TR} \leq \frac{(b - a)^3}{12N^2} M_2, \quad E_{SI} \leq \frac{(b - a)^5}{2880N^4} M_4 \]
- **Adaptive Simpson’s rule** uses error analysis/recursion
  - More efficient than composite methods, guarantees error
Algorithm: Composite Midpoint rule

- **Type:** Integral computing. Finds $\int_a^b f(x)dx$
- **Initialization:** $[a, b]$, number of intervals $N$
- **Mathematically:**
  \[
  \int_a^b f(x)dx \approx h \sum_{k=0}^{N-1} f\left(x_k + \frac{h}{2}\right), \quad h = \frac{a - b}{N}, \quad x_k = a + kh
  \]
- **Algorithm:**
  
  ```python
  h = (b-a)/N
  totalSum = 0
  for k in range(0,N):
    x_k = a + k*h
    totalSum += f(x_k + h/2)
  totalSum = h*totalSum
  return totalSum
  ```
Algorithm: Composite Trapezoidal rule

- **Type:** Integral computing. Finds $\int_{a}^{b} f(x)dx$
- **Initialization:** $[a, b]$, number of intervals $N$
- **Mathematically:**
  \[
  \int_{a}^{b} f(x)dx \approx \frac{h}{2} \left( f(x_0) + 2 \sum_{k=1}^{N-1} f(x_k) + f(x_N) \right),
  \quad h = \frac{a - b}{N}, \quad x_k = a + kh
  \]
- **Algorithm:**
  ```python
  h = (b-a)/N
  totalSum = f(a)
  for k in range(1,N):
    x_k = a + k*h
    totalSum += 2*f(x_k)
  totalSum += f(b)
  totalSum = h/2*totalSum
  return totalSum
  ```
Algorithm: Composite Simpson’s rule

- **Type:** Integral computing. Finds $\int_{a}^{b} f(x)dx$
- **Initialization:** $[a, b]$, number of intervals $N$
- **Mathematically:**
  \[
  \int_{a}^{b} f(x)dx \approx \frac{h}{3} \left( f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + \cdots + 2f(x_{2N-2}) + 4f(x_{2N-1}) + f(x_{2N}) \right)
  \]
  \[
  h = \frac{a - b}{2N}, \quad x_k = a + kh
  \]
- **Algorithm:**
  
  \[
  \begin{align*}
  h &= (b-a)/(2*N) \\
  totalSum &= f(a) \\
  for \ k in range(1,2N) \\
  x_k &= a + k*h \\
  if \ k \% 2 \ is \ 1: \ # \ Odd \ index \\
  totalSum &=+ 4*f(x_k) \\
  else: \ # \ Even \ index \\
  totalsum &=+ 2*f(x_k) \\
  totalSum &=+ f(b) \\
  totalSum &= h/3*totalSum \\
  return \ totalSum
  \end{align*}
  \]
Algorithm: **Adaptive Simpson’s rule**

- **Type:** Integral computing. Finds $\int_a^b f(x)\,dx$
- **Initialization:** $[a, b]$, error tolerance $\epsilon$
- **Algorithm:**
  
  ```python
  def ad_Simpson(f,a,b,eps):
      whole = Simpson(f,a,b)
      c = (a+b)/2
      left = Simpson(f,a,c)
      right = Simpson(f,b,c)
      if abs(whole - (left + right)) < 15*eps:  # Error OK
          return 16/15*(left + right) - 1/15*whole  # Extrapolation
      else:  # Error not OK, split interval in two
          return ad_Simpson(f,a,c,eps/2) + ad_Simpson(f,c,b,eps/2)
  ```
Week 40/41: Gaussian elimination

- **Task:** Solve a matrix-vector system $Ax = b$
- **The method:** Gaussian elimination + back substitution
- GE is a **direct** solver: Running the algorithm **gives the answer**, no iterations or error estimates
- Roundoff errors are minimized by **partial pivoting**
  - Swap rows such that the pivot element is maximal in its column
- After Gaussian elimination, use **back substitution** to find the answer
- Can be implemented **in-place**; don’t need to create new matrices, saves space
Algorithm: Gaussian elimination with partial pivoting

- **Type:** Linear equation solver. Solves: $Ax = b$
- **Initialization:** $N \times (N + 1)$ augmented matrix $M$
- **Pseudoalgorithm:**
  
  ```python
  row = 0, col = 0
  while (row < N-1 and col < N):
    ind_row_max = get_max(M,row,col) # Maximum in col
    if w_max][col] is 0: # Pivot element is 0
      col += 1 # No nonzero element in pivot column
    else:
      swap(M[row_ind],M[max_row_ind]) # Swap rows
      row_reduce(M,row,col) # Zero out rows below
    row += 1, col += 1
  x = back_substitute(M) # Back substitution
  ```
Week 42: Newton’s method in n-D

- **Task:** Solve $f(x) = 0$
- Very similar to the 1-D version, uses the Jacobian matrix

\[
J_f(y) = \begin{bmatrix}
\frac{\partial f_0}{\partial x_0}(y) & \frac{\partial f_0}{\partial x_1}(y) & \cdots & \frac{\partial f_0}{\partial x_n}(y) \\
\frac{\partial f_1}{\partial x_0}(y) & \frac{\partial f_1}{\partial x_1}(y) & \cdots & \frac{\partial f_1}{\partial x_n}(y) \\
\vdots & \vdots & \ddots & \vdots \\
\frac{\partial f_n}{\partial x_0}(y) & \frac{\partial f_n}{\partial x_1}(y) & \cdots & \frac{\partial f_n}{\partial x_n}(y)
\end{bmatrix}
\]

1. Solve the linear system $J_f(x^k) z = -f(x^k)$
2. Compute $x^{k+1} = x^k + z$

- Stopping conditions must take all dimensions into account
  - Example: $|f_0(x^k)| < \varepsilon$ and $|f_1(x^k)| < \varepsilon$ … and $|f_n(x^k)| < \varepsilon$, and/or $|x^k_0 - x^{k-1}_0| < \delta$ and … and $|x^k_n - x^{k-1}_n| < \delta$. 
Algorithm: Newton’s method in n-D

- **Type:** Equation solver. Finds zeroes: \( f(x) = 0 \)
- **Initialization:** \( x^0 \), tolerances \( \epsilon, \delta \).
- **Mathematically:**
  - Solve the linear system \( J_f(x^k)z = -f(x^k) \)
  - Compute \( x^{k+1} = x^k + z \)
- **Pseudoalgorithm:**
  
  \[
  k = 0 \\
  \text{while } \langle \text{stopping conditions are not satisfied} \rangle \\
  \quad \text{compute } J_f(x^k), \ f(x^k) \\
  \quad \text{solve the linear system } J_f(x^k)z = -f(x^k) \\
  \quad x^{k+1} = x^k + z \\
  \quad k += 1 \\
  \text{return } x^k
  \]
Week 43/44: Methods for solving ODEs

- **Task**: Solve the ODE \( \dot{x}(t) = f(x, t) \); solution is \( x(t) \)
- **Numerically**: find a series \( \{x^k\}_{k=0}^N \), \( x^k \approx x(kh) \)
- Formulation of methods is the same for 1-D and n-D
- Methods can be **explicit** or **implicit**
  - Explicit: \( x^k \) can be computed directly (explicit Euler, Heun’s)
  - Implicit: \( x^k \) is computed by solving an equation (implicit Euler)
- Methods can have several **stages**
  - Combine several estimates of the slope to get a better fit.
    - Heun’s method is a 2-stage method
- **Stability**
  - A method is unstable if \( x^k \to \infty \) as \( k \to \infty \) when applied to the test equation \( f(x, t) = -\lambda x, \quad \lambda \geq 0 \)
  - Implicit methods are often more stable but slower than explicit methods
- **Convergence order**
  - A method is of order \( p \) if \( |x^k - x(kh)| < C_k h^p \)
  - An order \( p \) method improves its answer by a factor \( 2^p \) when \( h \to h/2 \)
  - Explicit/Implicit Euler are order 1, Heun’s method order 2
Algorithm: **Explicit Euler**

- **Type**: ODE solvers. $\dot{x}(t) = f(x, t), \ x(0) = x^0$
- **Initialization**: $x^0, T, N$
- **Mathematically**: $x^{j+1} = x^j + hf(x^j, t_j)$
- **Pseudoalgorithm**:

```python
x_list = [x^0]
x = x^0
h = T/N
for j in range(N):
x = x + hf(x, jh)
x_list.append(x)
return x_list
```
Algorithm: Implicit Euler

- **Type:** ODE solvers. $\dot{x}(t) = f(x, t)$, $x(0) = x^0$
- **Initialization:** $x^0, T, N$
- **Mathematically:** $x^{j+1} = x^j + hf(x^{j+1}, t_{j+1})$
- **Pseudoalgorithm:**

```python
x_list = [x^0]
x = x^0
h = T/N
for j in range(N):
    solve the equation $y = x + hf(y, (j+1)h)$
    x = y
    x_list.append(x)
return x_list
```
**Algorithm: Heun’s method**

- **Type:** ODE solvers. \( \dot{x}(t) = f(x,t), \ x(0) = x^0 \)
- **Initialization:** \( x^0, T, N \)
- **Mathematically:**
  
  \[
  s^{j+1} = x^j + hf(x^j, t_j)
  \]
  
  \[
  x^{j+1} = x^j + \frac{h}{2} \left( f(x^j, t_j) + f(s^{j+1}, t_{j+1}) \right)
  \]

- **Pseudoalgorithm:**

  ```python
  x_list = [x^0]
  x = x^0
  h = T/N
  for j in range(N):
    s = x + hf(x, jh)
    x = x + h/2*(f(x, jh) + f(s, (j+1)h))
    x_list.append(x)
  return x_list
  ```
Week 41: Plotting

• Include matplotlib using the command
  ```python
  import matplotlib.pyplot as plt
  ```

• Given lists $x$ and $y$ of equal length, we plot the points $(x[i], y[i])$ with the command `plt.plot(x, y)`
  – Same as when drawing a graph from hand if you have no idea how it looks: put dots on the coordinates and draw lines between

• To see the figure, use `plt.show()`

  #Inform about label on the y axis
  ```python
  plt.ylabel('some numbers')
  ```

  #Axis range: $[x_{\text{min}}, x_{\text{max}}, y_{\text{min}}, y_{\text{max}}]$
  ```python
  plt.axis([0, 4, 0, 16])
  ```
Plotting styles

• The default behaviour of `plt.plot()` is to connect the points with lines
• We can change this using additional arguments after the x/y coordinates
  – For example, to plot y over the x points as red circles:
    `plt.plot(x, y, 'ro')`
  – To plot y over the x points as green triangles:
    `plt.plot(x, y, 'g^')`
Plotting several graphs in one figure

- If we want to generate several graphs, plot all of them first using `plt.plot()`, then use `plt.show()`

```python
#Import plotting library
import matplotlib.pyplot as plt
x = ...
y1 = f(x)
y2 = g(x)
plt.plot(x,y1)
plt.plot(x,y2)
plt.show()
```
Questions?