Titre: Note sur le fichier Ti “ELE735\_t4.tns”

Par: Vincent Rougeau-Moss, le programmeur de ce fichier Ti

Fichier Ti construit durant le cours d’ELE735 de l’automne 2016

Date production: 16 au 18 décembre 2016, après l’examen final

Date dernière mise à jour après production: 20 septembre 2017

Table des matières

[**Introduction**](#_cn0ecatz43jx) **2**

[**Fonctionnement**](#_3rmnl61dj61s) **4**

[**Guide de l’utilisateur du fichier Ti**](#_1rfts2yu5u8a) **5**

[**Activité 1: Équation et Système d’Équation Non-Linéaire + Méthode de résolution de système linéaire par méthode récursive.**](#_8a1ds2yl010f) **6**

[Méthode de la Bisection:](#_reou8w5dm9lv) 6

[Méthode par Interpolation linéaire:](#_dc85ghu6zd3r) 7

[Méthode de Newton:](#_br83kry7xf75) 8

[Méthode de la Sécante (méthode du prof) (ne pas faire celle du manuel à l’examen [j’avais fait les deux et il a barré celle du manuelle + enlever des points]):](#_uear7lkf7lj9) 9

[Méthode de la sécante (méthode du manuel):](#_ql4ihdmt6nz) 10

[Méthode du point fixe:](#_uyt8ovp4wvs7) 10

[Résolution de système d’équations Non-Linéaire par la méthode de Newton:](#_14zhrwvj0n2h) 11

[Méthode de résolution de système linéaire par méthode itérative de Jacobi - version Matricielle:](#_dlc3on5k6svj) 12

[Méthode de résolution de système linéaire par méthode itérative de Jacobi (méthode itérative) et de Gauss-Seidel:](#_63xl04i745yf) 14

[**Activité 2: Résolution de système linéaire par méthode Matricielle (sauf Jacobie matricielle), SVD, eigenvalue-eigenvector et vérification pour savoir si matrice orthogonale**](#_vxoid0misdl5) **15**

[Méthode de Gauss:](#_j8mwyxr8igw1) 15

[Méthode de Gauss-Jordan:](#_zas1cg1u1een) 16

[Méthode de résolution par la décomposition LU:](#_pivil9rxriic) 17

[Valeurs propres (Eigenvalues):](#_8t511wsibkew) 18

[Décomposition en Valeurs Singulières - SVD (Singulars Values Decomposition)](#_3h95tdw07ugg) 19

[Matrice inverse ou pseudo-inverse (si rectangulaire) à l’aide de la SVD](#_7vwhyssmwrvx) 20

[Vérification pour savoir si une matrice est orthogonale](#_uazo7qxmbzt8) 20

[**Activité 3: Régression et Interpolation**](#_tmauxryuecy) **20**

[Régression Linéaire d’équation Linéaire ou Non-Linéaire (qui ont été mis sous la forme linéaire)](#_l5g5gm3bhg2w) 22

[Régression polynomiale](#_kfsrnbvmdljj) 22

[Interpolation Polynomiale - Méthode la plus simple](#_qcmy9i6wvt7y) 24

[Interpolation par polynôme de Lagrange](#_dd00ywyw69d0) 24

[Interpolation par polynôme de Newton:](#_bushkh2gine5) 25

[Introduction aux splines](#_8qba7998yp73) 26

[Spline Linéaire (ordre 1)](#_w043uxnz230z) 27

[Spline Quadratique (ordre 2)](#_rfdgqciqeago) 27

[Spline Cubique (ordre 3)](#_jx881ikaiad7) 29

[**Activité 4: Dérivé Numérique**](#_ytyy6u0a6f8) **30**

[Dérivé première numérique](#_tyxq7dyo5x94) 31

[Dérivé seconde numérique](#_luh2ka253n73) 31

[Dérivé tertiaire numérique](#_rxqj9rdscxmr) 31

[Dérivé 4ème numérique](#_tc9oma4kcb38) 32

[Plusieurs dérivées de suite](#_v43o5q8tdf9h) 32

[**Activité 5: Intégration numérique**](#_puwk1y6gnld0) **33**

[**Activité 6: EDO et Système d’EDO d’ordre 1 à une ou plusieurs variables (ordre supérieur par encore fait)**](#_644ssuup8jbm) **34**

[EDO ordre 1 à 1 variable - Méthode d’Euler et du Point Milieu (Midpt)](#_vogxvb2dxfre) 35

[EDO ordre 1 à 1 variable - Méthode de Runge-Kutta](#_6wzyyiya34t6) 35

[Système d’EDO d’ordre 1 - Méthode d’Euler ou du Point Milieu (Midpt)](#_2e1fbpdv67p2) 36

[Système D’EDO d’ordre 1 - Méthode de Runge-Kutta](#_ikqd21488407) 36

# Introduction

Le cours d’ELE735, Analyse Numérique, porte sur différents algorithmes de calcul mathématique qui permettent de faire des calculs numériques. Comme il est expliqué dans le premier cours, le plus souvent, lorsqu’on fait du calcul analytique (ex.: y = exp(x+sin(x))), on va approximer la situation afin de la simplifier, car sinon il serait impossible de la calculer, ou très difficile. On peut notamment penser au cours de physique du secondaire et du cégep sur les projectiles avec y\_f = y\_i + Viy\*t -½\*g\*t^2. On prend seulement en considération la gravité et on néglige plusieurs autres facteurs tels que l’aérodynamisme, la résistance de l’air, la force de gravité en fonction de l’altitude, l’effet de Coriolis, etc. Si l’on veut prendre en compte un grand nombre de facteurs, on doit soit résoudre des calculs complexes uniquement analytiquement, soit passer par des calculs numériques qui vont permettre d’approximer la solution avec une certaine marge d’erreur, marge d’erreur qui comprend celle de l’ordinateur (au niveau binaire) ainsi que la marge d’erreur du calcul.

Au cours du cours, plusieurs algorithmes ont été présentés et l’ensemble de ces méthodes mathématiques ont été programmés dans la calculatrice Ti nspire cx CAS à l’ordinateur. Tous ont été testés, vérifiés, corrigés et vérifier à nouveau à l’aide de tous les exercices à faire (100 %), les 4 tests, les TP et même l’examen final. J’ai eu une confiance absolue par rapport à mes programmes, principalement par le fait que :

1. Manque de temps durant les examens → Pas le temps de le faire à la main sans faire d’erreur d'inattention et pas le temps de réviser ces calculs durant les examens.
2. Perds 0.5/10 par fautes durant les tests (soit 5% par fautes de calculs) qui compte au total pour 40 % (10% par test) de la note final (à vérifier).
3. Les méthodes sont des méthodes numériques pour ordinateurs → Pourquoi les faire à la main quand tu as un ordinateur avec toi ?
4. J’ai préféré passer 1 à 3h par fonction/programme pour m’assurer d’avoir une méthode parfaite que je peux appliquer des milliers de fois en quelques secondes au lieu de devoir à chaque fois le faire à la main en quelques minutes.
5. Le code est principalement basé sur les méthodes présentées durant le cours, les TP ou le manuel.
6. Toutes les fonctions et tous les programmes donnent leur démarche complète (comme si on l’avait fait à la main) et qu’on a juste besoin de le retranscrire sur la feuille d’examen.

Le fichier final qui à été utilisé durant l’examen, soit celui sur le Drive, pèse 170 Ko. Durant l’examen, je n’ai eu aucun problème ni aucune attente à faire ou d’horloge qui tourne malgré tout. Le truc: Séparé chaque section à l’aide d’activité. En effet, les activités d’un même fichier sont toutes indépendantes entre elle et donc, les noms de fonctions et les variables de l’activité 1 (par exemple) n’existent pas dans les autres activités. Le seul problème est qu’effacer quelque chose ou enregistrer le fichier prend quelques secondes. Brancher la Ti semble peut-être l’aider à être plus rapide.

Le fichier, avant réduction pour l’examen final, pesait 210 Ko. Tout ce que j’ai fait, c’est enlever les fonctions en double (les versions 1), enlever les fonctions de la page graphique des méthodes de régressions et d’interpolation, supprimer cette page graphique, fermer la page du code de la spline cubique (à libérer 10 Ko simplement en fermant la page du code sans la détruire) et réduire les exemples d’exécution à moins d’itération afin d’avoir des exemples qui prennent moins de ligne tout en gardant un exemple d’exécution, puisqu’à l’examen, je veux juste savoir ce qu’elle fait et non testé sa performance.

Durant la préparation à l’examen final, presque toutes les méthodes qui avaient été faites dans la Ti ont été refaites en Matlab. Les parties qui manquaient (en date du 18 décembre 2016) ont été ajoutées par la suite dans Matlab, mais n’ont pas encore été confirmées ni vérifiées avec des exercices. En fait (en date du 11 mai 2017), le code n’a pas été encore testé avec des exercices, mais a généralement été testé (avec un cas) entre la Ti et Matlab. Puisque la vérification n’est pas encore faite (et donc que des changements sont possible), la version Matlab n’a pas encore été publiée. Dans le cas où quelqu’un voudrait malgré tout y avoir accès, demandé à Vincent Rougeau-Moss.

# Fonctionnement

Le fichier est décomposé en 6 activités, chacune se rapportant à un sujet ou un test durant le cours. Pour chaque activité, la première page est une page de calcul pour pouvoir exécuter les fonctions (notamment à l’examen) et il y a une autre page de calcul d’exemple d’exécution avec chacune des fonctions, ce qui permet à l’examen de simplement copier-coller l’en-tête, modifier un peu si besoin les paramètres et cela permet d’avoir une idée de ce qu’elle va afficher.

Toutes les fonctions font soit dire la démarche qu’elles ont faite étape par étape, ce qui doit être retranscrit sur la feuille d’examen, soit dire que le calcul qu’ils ont fait (parce que c’est une ligne de code) est présenté à tel numéro de page dans le manuel. C’est notamment le cas avec les 6 méthodes des 4 dérivés numériques (“diff\_no”).

Certaines fonctions, comme celle des EDO, font en plus dire à l’utilisateur ce qu’il a entré, afin que l’utilisateur puisse s’assurer qu’il a rentré les bons paramètres et corrigé au besoin par la suite. Ceci a été ajouté dû à la grande quantité de paramètres à mettre et une généralisation pour certains cas a été ajoutée.

Toutes les fonctions, sauf les fonctions de dérivée numérique “diff\_no”, ont des vérifications des paramètres qui font avertir l’utilisateur s’il a mit les mauvais paramètres en lui disant ce qu’il devait mettre. Cela évite d’avoir besoin d’aller voir le fichier de code pour savoir quoi mettre, ce qui est très utile si on manque de temps à l’examen. De plus, cela évite d’avoir un message d’erreur durant l’examen. Ainsi, si tu ne sais pas quoi mettre, met (exemple) 1 paramètre random → Échec, car pas assez de paramètres, tu en rajoute un et tu continues ainsi jusqu’à avoir un message du programme qui te dit que tu n’as pas rentré les bons paramètres et qui te dis ce que tu devais mettre.

Un document de guide d’utilisation avait été fait avant l’examen, mais uniquement de manière écrite sur papier. Ce document était de seulement 3 pages recto verso. Une version Matlab du programme Ti a été commencée avant l’examen et contient tout, sauf les splines (en date du 16 décembre 2016)

# Guide de l’utilisateur du fichier Ti

La très longue liste suivante explique en détail le fonctionnement de chacune des fonctions, les paramètres à mettre et, la plupart du temps, plus de détail afin d’aider les utilisateurs qui n’ont pas fait le cours (notamment pour certaines connaissances particulières). Ainsi, la version initiale (qui était papier) avait 1 page recto par activité, pour un total de 3 pages recto verso. Le document ici est beaucoup plus long pour plusieurs raisons telles que le fait que c’est en liste (texte pas comprimé) et le fait qu’il y a beaucoup d’explication sur le fonctionnement de chacune de ces méthodes de calcul numérique et sur comment utiliser chacune de ces méthodes. Au besoin, télécharger le document et enlever le texte d’explication pour une version comprimé (ex.: pour l’examen). Sinon, faites juste copier-coller la section des paramètres à mettre pour chacune des méthodes et enlever les commentaires supplémentaires d’explications.

Toutes les fonctions et tous les programmes, sauf celle “diff\_no” de l’activité 4, ont des vérifications du type des paramètres qui font donner un message d’erreur avec une explication de ce qu’il faut mettre si les paramètres sont invalides.

Pour les fonctions et les programmes des activités 3 et suivantes, une fonctionnalité supplémentaire de conversion de vecteur en liste a été ajoutée (pour la plupart des fonctions et programmes). Ainsi, même si le paramètre idéal est une liste, la fonction est capable de transformer le vecteur en liste au besoin, mais des opérations supplémentaires seront nécessaires (d’où pourquoi le guide va dire de prendre une liste et non un vecteur).

Garder aussi en tête que par le fait que c’est des méthodes numériques, il est possible que les algorithmes divergent et donc, donnent des résultats extravagants et vraiment mauvais. C’EST NORMAL ! C’est la méthode qui est faite comme cela et comme n’importe quelle méthode numérique, elles ont chacune des faiblesses et des conditions qui font en sorte que parfois cela va diverger ou parfois cela ne sera pas applicable. Une notion importante que tout ingénieur doit garder en tête c’est que les simulateurs utilisent aussi ce même genre d’algorithme et eux aussi peuvent diverger. Ne jamais se fier aveuglément à un simulateur (notion importante acquise dans le cours). Le guide contient certaines spécificités par rapport aux conditions de divergences (spécificité qui vient soit du cours, soit du manuel, dans la plupart des cas).

# Activité 1: Équation et Système d’Équation Non-Linéaire + Méthode de résolution de système linéaire par méthode récursive.

* + Toutes les méthodes, sauf celle du point fixe, fonctionnent sous le principe que la fonction doit être exprimé sous la forme f(x) = 0 dans lequel on envoie “f(x)” à la fonction de calcul (sans le “= 0”). Le but est alors de trouver la valeur de X permettant d’avoir f(x) = 0. La valeur obtenue sera une approximation de la valeur réelle de X.
  + L’approximation sera meilleure si l’erreur maximale exigée est plus petite et si le nombre de boucles à faire est plus grand.
  + Par défaut, toutes les fonctions ont été mises pour faire maximum 10 boucles, puisqu’on n’a jamais eu besoin de faire plus qu’environ 5 boucles à la main.
  + Il y a deux types de méthodes dans celle qui suit:
    - 1) Méthode par intervalle: On a deux points autour de la solution (autour du x pour lequel f(x) = 0) et on réduit l’intervalle à mesure qu’on fait des itérations. Méthode plus lente que le second type, mais plus fiable.
    - 2) Méthode qui suit la fonction: On a un point de la fonction et à l’aide des valeurs de la fonction et de sa dérivée première, on converge vers la solution. Méthode plus rapide, mais qui diverge plus facilement.
  + Un point important à garder en tête est que toutes ces méthodes convergent plus facilement et plus vite si l’intervalle est faible ou si le point de départ est proche de la solution. Plus on est loin de la solution, plus il y a un risque important de diverger.

## Méthode de la Bisection:

* + - La méthode de la bissectrice est la méthode la plus fiable, puisqu’elle converge toujours, mais c’est la plus lente des méthodes.
    - La méthode fonctionne en utilisant uniquement la valeur des fonctions au point de son intervalle et n’utilise donc pas la forme de la fonction pour identifier la solution. Ceci lui permet de ne jamais diverger, mais nuit dans sa performance.
    - Le principe est que f(xl)\*f(xh) <= 0 si la solution se trouve entre xl et xh. On va donc à chaque itération calculer le point milieu de cet intervalle (x\_milieu) et choisir entre l’intervalle [xl, x\_milieu] et [x\_mileu, xh]. Le choix se fait de manière à prendre l’intervalle où la solution se trouve. Donc, on prend [x\_milieu, xh] si f(x\_milieu)\*f(xh) <= 0 et si faux, on prend l’intervalle [xl, x\_milieu].
    - Une fois le nouvel intervalle choisit on change xl ou xh afin de prendre ce nouvel intervalle pour l’itération suivante.
    - Dans le cas où la performance n’est pas un problème trop important (en termes de temps) et que vous avez peu d’information sur où se trouve la solution, cette méthode est pratique grâce au fait qu’elle va toujours converger (si bien sûr la solution se trouve dans l’intervalle de départ qu’on fournit à la fonction).
    - **metbisection (fonction, xl, xh, emax)**
      * fonction: f(x) exprimé sous la forme: f(x) = 0
      * xl = début de l’intervalle en X (où le x pour f(x) = 0 se trouve)
      * xh = fin de l’intervalle en X (où le x pour f(x) = 0 se trouve)
      * Arrêt de la boucle lorsque :
        + |xl - xh| <= emax
        + Plus que 10 boucles (mis par défaut, car on ne fera jamais 10 boucles à la main à l’examen)

## Méthode par Interpolation linéaire:

* + - La méthode fonctionne sous le principe que f(xl)\*f(xh) < 0 si le la solution se trouve dans l’intervalle [xl, xh] (*fait de mémoire*).
    - À chaque itération, on calcule le point milieu de cet intervalle et on remplace le point xl ou xh en fonction de la plus grande valeur (en valeur absolue) entre f(xl) et f(xh), ce qui permet de réduire l’intervalle au fur et à mesure qu’on avance.
    - Cette méthode fonctionne très bien si la fonction est linéaire sur l’intervalle [xl, xh], fait va plus facilement diverger si l'intervalle [xl, xh] est composé d’une courbe très prononcée (polynômes d’ordre 2) ou s’il y a plusieurs courbes.
    - Cette méthode converge plus vite que la méthode de la bissectrice, mais peut diverger.
    - **metinlin(fonction, xl, xh, emax)**
      * fonction: f(x) exprimé sous la forme: f(x) = 0
      * xl = début de l’intervalle en X (où le x pour f(x) = 0 se trouve)
      * xh = fin de l’intervalle en X (où le x pour f(x) = 0 se trouve)
      * Arrêt de la boucle lorsque :
        + |xl - xh| <= emax
        + Plus que 10 boucles (mis par défaut, car on ne fera jamais 10 boucles à la main à l’examen)

## Méthode de Newton:

* + - La méthode de Newton est une méthode du type (suit la fonction) qui, lorsqu’elle ne diverge pas, converge très rapidement.
    - Cette méthode utilise le calcul suivant à chaque itération: [https://lh5.googleusercontent.com/vx2anEmr7UuL-sGJpTqLqvPPPV2M7s0NoU26jEFGyLi2zoQDjRpXO_OBQxsHmtK74lsNPL5tT1m9whyqmVnt7EPruN3XHiONfx-jwim-EqBW1MTWDttMm0fjCOnU-Z64IczYXI4X](http://api.gmath.guru/cgi-bin/gmath?%5Cdpi%7B480%7Dx_%7Bsuivant%7D%5C%20%3D%5C%20x_i%5C%20-%5Cfrac%7Bf%5Cleft(xi%5Cright)%7D%7Bf%27%5Cleft(x_i%5Cright)%7D)où la fin du calcul xi devient x suivant.
    - Cette méthode a cependant des critères importants pour être utilisée et pour aider à ne pas diverger (aucune vérification à ce niveau dans la Ti):
      * 1) La fonction est continue autour de la solution et au point de départ fourni.
      * 2) La dérivée de la fonction existe et est continue autour de la solution et au point de départ fourni
      * 3) La solution ne se trouve pas sur un sommet → la dérivée de la fonction autour et à la solution != 0.
    - L’autre point important avec cette méthode est qu’on doit soit savoir la fonction de la dérivée, soit calculer manuellement la dérive à chaque itération, ce qui n’est pas très performant. Actuellement, celle de la Ti calcule à chaque fois la dérivée et évalue celle-ci par rapport au point fourni (serait à améliorer).
    - Si la dérivée est inconnue, la méthode de la sécante fait une approximation de la dérivée.
    - Lorsque la méthode de Newton diverge, elle diverge habituellement complètement (des valeurs gigantesques sont obtenues), on sait donc facilement si elle a divergé puisque les valeurs obtenues après chaque itération ne font qu’augmenter (en valeur absolue).
    - **metnewton (fonction, xi, emax)**
      * fonction: f(x) exprimé sous la forme: f(x) = 0. Important: la dérivée de la fonction et la fonction doivent être continues autour de la solution.
      * xi = valeur de départ de la recherche (doit être proche de la solution, soit le x qui permet f(x) = 0)
      * Arrêt de la boucle lorsque:
        + |(x\_suivant-xi)/xi| <= emax (ce qu’on appelle dans le manuel la “tolérance”)
        + Plus que 10 boucles (mis par défaut, car on ne fera jamais 10 boucles à la main à l’examen

## Méthode de la Sécante (méthode du prof) (ne pas faire celle du manuel à l’examen [j’avais fait les deux et il a barré celle du manuelle + enlever des points]):

* + - La méthode de la sécante est exactement pareille à la méthode de Newton, à la différence qu’au lieu de devoir connaître la dérivée (ou la calculer analytiquement, comme avec la Ti), on va approximer à l’aide de la méthode de 2 points suivants (diff\_1, choix = 1 dans activité 4).
    - La différence entre la méthode du prof et la méthode du manuel est que la méthode du prof va utiliser l’espacement entre les points (qui sera considéré comme étant constant) afin de faire le calcul. Celle du manuel ne va pas faire cette approximation. Les deux sont théoriquement équivalentes, car celle du prof est basée à la base sur celle du manuel.
    - **methsecant (fonction, xi, ઠ, emax)**:
      * fonction: f(x) exprimé sous la forme: f(x) = 0
      * xi = valeur de départ de la recherche (doit être proche de la solution, soit le x qui permet f(x) = 0)
      * ઠ : l’intervalle entre les valeurs en X où plus spécifiquement: x[i] - x[i-1] = xh - xl
      * Arrêt de la boucle lorsque:
        + |(x\_suivant-xi)/xi| <= emax
        + Plus que 10 boucles (mis par défaut, car on ne fera jamais 10 ne boucle à la main à l’examen)

## Méthode de la sécante (méthode du manuel):

* + - La méthode originale sur lequel celle du prof s’est basée. Au lieu d’avoir un seul point et d’utiliser l’espacement entre les points, on va directement définir soit même son espacement via deux points.
    - La méthode à la base avait été faite suite au fait que celle du prof ne donnait jamais les mêmes résultats que le corrigé dans les exercices. Suite au fait que durant le 1er test, alors que j’avais mis les résultats obtenus par les deux méthodes, le prof avait barré celle du manuel et m’avait enlevé des points, j’ai revu les documents de cours et je me suis assuré que celle de la Ti soit exactement comme dans son PowerPoint. Cependant, je n’ai pas revérifié après cela pour savoir si j’obtenais dorénavant les bons résultats avec la méthode du prof.
    - **metsecante(fonction, x1, x2, emax)**
      * fonction: f(x) exprimé sous la forme: f(x) = 0
      * x1 et x2 sont les bornes de l’intervalle [x1, x2] autour duquel se trouve la solution, soit la valeur en x pour lequel f(x) = 0.
      * Arrêt de la boucle lorsque:
        + |(x\_suivant-xi)/xi| <= emax
        + Plus que 10 boucles (mis par défaut, car on ne fera jamais 10 boucles à la main à l’examen)

## Méthode du point fixe:

* + - La méthode du point fixe est la seule méthode qui n’utilise pas la forme “f(x) = 0”, mais plutôt la forme “f(x) = 0 → x = g(x)”. Cette méthode a aussi amener à des méthodes de résolution de système linéaire par méthode itérative de Jacobi et de Gauss-Seidel qui sont basés sur cette méthode.
    - Pour exprimer la fonction “f(x) = 0” sous la forme “x = g(x)”, il y a 2 méthodes:
      * 1) Si on peut facilement isoler l’un des X de la fonction, il suffit alors d’isoler l’un des X afin d’obtenir une nouvelle expression qui sera “g(x)”. Dans la Ti, la méthode la plus simple est de juste remplacer chacun des x par une autre lettre ou mettre un numéro à côté de chacun des x puis faire solve en fonction de chacun de chacun de ces x. Vous obtiendrez alors plusieurs formes de “g(x)” que vous pourrez tester. Certaine ne pourront pas converger (la fonction vérifie cela) et d’autre oui.
      * 2) Si on ne peut pas facilement l’un des X (ex.: cos(x) = 0 ou sin(x) = 0). Alors on a qu’à exprimer que g(x) = f(x) + x (*de mémoire*).
    - Cette méthode à l’avantage de pouvoir être utilisé pour d’autre cas que ceux où on peut facilement exprimer la fonction sous la forme “f(x) = 0” et à l’avantage qu’on a un critère qui permet de savoir si elle va converger.
    - Le critère de convergence est : g’(x) < 1 autour de la solution. La Ti fait donc le calcul de la dérivée et l’évaluent par rapport au point de départ pour savoir si elle va diverger.
    - Dans le cas d’une implémentation numérique, on peut utiliser une approximation de la dérivée première pour faire cela.
    - **metptfixe (fonction, xi, max\_loop)**
      * fonction: f(x) exprimé sous la forme x = g(x) et si pas capable, mettre g(x) = f(x) + x. Si plus d’une possibilité de x pouvant être isolé, toutes les essayer, car certaine ne font pas répondre aux critères de convergence, ce qui est vérifié par la fonction (ne le fait pas si va diverger)
      * xi = valeur du point de départ proche de la solution
      * max\_loop = nombre de boucles à faire.

## Résolution de système d’équations Non-Linéaire par la méthode de Newton:

* + - Note importante: Nécessite le fichier de Vincent Rougeau-Moss: “Pubfonct.tns” qui contient une méthode pour faire le “Jacobian”, une matrice de dérivée partielle des fonctions en fonction de chaque variable. Vous ne pourrez pas exécuter ce programme sans cela.
    - La méthode de résolution de système d’équations non linéaires par Newton est une généralisation de la méthode de Newton par rapport à un système de fonction. Elle a donc les mêmes critères et les mêmes forces et faiblesses que la méthode de Newton, mais due au fait que les critères sont à appliquer pour toutes les fonctions, ce n’est pas toujours utilisable.
    - En effet, il faut que toutes les conditions suivantes soient respectées:
      * Toutes les fonctions existent et sont continues autour des solutions (autour des variables permettant d’obtenir les solutions)
      * Toutes les fonctions sont dérivables et leurs dérivées existent et sont continues autour des solutions.
      * Le Jacobian (une matrice des dérivées partielles de toutes les fonctions en fonction de toutes les variables) != 0 autour des solutions.
      * La matrice inverse de la matrice Jacobienne existe toujours autour des solutions.
      * La matrice de fonction doit être carrée, donc nombre de fonctions = nombre de variables.
    - Cette méthode peut converger très vite ou diverger très vite, ce qui est aussi le cas avec la méthode de Newton.
    - **metnewtonsys(vecteur\_fonctions, liste\_variables, vector\_ci, max\_loop)**
      * vecteur\_fonction: vecteur de N fonction de plusieurs variables qui sont toute sont la forme f(x,y, …) = 0. Important: toutes les fonctions doivent toutes être dérivables et continue autour de la solution.
      * liste\_variables: liste des variables dans le même ordre que vector\_ci.
      * vector\_ci : vecteur des valeurs de départ de toutes les variables (ci = Condtion Initiale). L’ordre doit être le même que pour liste\_variables.
      * max\_loop : nombre de boucles à faire.

## Méthode de résolution de système linéaire par méthode itérative de Jacobi - version Matricielle:

* + - La méthode de Jacobi est une méthode assez simple qui est tout simplement basée sur la méthode du point fixe dans lequel on a isolé l’un des X de l’équation linéaire.
    - Par exemple, l’équation : a\*x1 + b\*x2 + c\*x4 + d\*x5 = y1, peut être mis sous 5 formes différentes, une pour chacun des X. C’est le principe que la méthode de Jacobi utilise. À chaque itération, on calcule la valeur de l’un des X et une fois le calcul de tous les X fait, on remplace les valeurs de chacun des X par sa valeur suivante (celle qui a été calculée).
    - Par le fait qu’on attend d’avoir terminé de calculer tous les X avant de changer leurs valeurs, il a été possible de l'exprimer sous la forme matricielle, ce qui permet, lorsqu’on a logicielle ou un programme de calcul, de faire une seule boucle pour les “nb\_loop” à faire au lieu de 3 boucle imbriquée.
    - Le programme “meth\_jacobi1” utilise la méthode matricielle tandis que celle dans “gaussiedel” utilise la méthode sans matrice, celle originale. À noter que le manuel ne donne que la méthode originale et que c’est le prof qui à exprimer cette méthode sous une forme matricielle.
    - Un des critères (non obligatoire) pour aider à la convergence est que la diagonale de la matrice A (dans A\*x = B où x = ?) est dominante, donc la valeur du coefficient sur la diagonale de chacune des lignes est toujours supérieur à la somme des autres coefficients de la ligne. Ceci n’est pas obligatoire à la convergence, mais elle ne va pas diverger si cette condition est respectée.
    - **meth\_jacobi1(matrice, solution, xi, nb\_loop)**
      * matrice: la matrice des équations exprimées sous une forme linéaire. Dans le calcul matriciel A\*x = B où x = ?, matrice = A.
      * solution: le vecteur colonne des valeurs auquel doivent valoir les équations linéaires. Dans le calcul matriciel A\*x = B où x = ?, solution = B.
      * xi : vecteur des valeurs de départ que doivent valoir les variables. Si vous n’avez aucune valeur de départ, mettre un vecteur de 0.
      * nb\_loop: le nombre de boucles à faire (car c’est une méthode itérative).

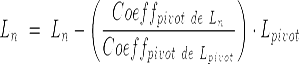
## Méthode de résolution de système linéaire par méthode itérative de Jacobi (méthode itérative) et de Gauss-Seidel:

* + - Le programme suivant de la Ti contient 2 méthodes itératives. La première est la forme originale de la méthode de Jacobi, qui utilise 3 boucles imbriquées, et la 2ème est la méthode de Gauss-Seidel.
    - La méthode de Gauss-Seidel est pareille à la méthode itérative de Jacobie à la seule différence qu’au lieu de devoir attendre d’avoir obtenu toutes les valeurs des X avant de faire une mise à jour, on le fait dès qu’on a obtenu une nouvelle valeur.
    - Cette méthode converge donc beaucoup plus vite que celle de Jacobi, mais ne peut être exprimée sous la forme de matrice (ce qui ne change rien si vous voulez la mettre dans un programme ou un langage sans matrice déjà existante).
    - À noter que les exercices du manuel utilisent toujours celle de Gauss-Seidel et jamais celle de Jacobi (même s’il dise utiliser Jacobi). Ceci peut se voir par la méthode de calcul utilisé qui est souvent exprimé sous la forme de code Matlab. En effet, pour faire la méthode Jacobi on doit utiliser une ou plusieurs variables temporaires pour enregistrer les nouvelles valeurs afin de les assigner au X à la fin. La méthode de Gauss-Seidel ne va pas utiliser de variable temporaire et va directement mettre les nouvelles valeurs à la place de l’ancienne.
    - Le même critère (non obligatoire) exposé dans la méthode de Jacobi matricielle s’applique toujours pour les deux méthodes itératives.
    - **gausssiedel(matrice, solution, xi, nb\_loop)**:
      * Ce programme fait à la fois la méthode de Jacobi (version itérative présentée dans le manuel) et la méthode de Gauss-Seidel. Une question sera posée à l'utilisateur pour cela, si les paramètres sont valides.
      * La différence entre les deux est que Gauss-Seidel converge beaucoup plus vite dû au fait qu’il met à jour les valeurs des variables dès qu’il obtient une nouvelle valeur au lieu d’attendre d’avoir toutes les valeurs des variables (comme la méthode matricielle de Jacobi).
      * matrice: la matrice des équations exprimées sous une forme linéaire. Dans le calcul matriciel A\*x = B où x = ?, matrice = A.
      * solution: le vecteur colonne des valeurs auquel doivent valoir les équations linéaires. Dans le calcul matriciel A\*x = B où x = ?, solution = B.
      * xi : vecteur des valeurs de départ que doivent valoir les variables. Si vous n’avez aucune valeur de départ, mettre un vecteur de 0.
      * nb\_loop: le nombre de boucles à faire (car c’est une méthode itérative).

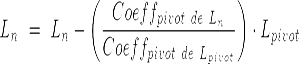
# Activité 2: Résolution de système linéaire par méthode Matricielle (sauf Jacobie matricielle), SVD, eigenvalue-eigenvector et vérification pour savoir si matrice orthogonale

* + Les méthodes de résolution de système linéaire sont un aspect assez important, car bien que la grande majorité des équations en ingénierie sont non linéaires (lorsqu’on ne les approxime pas), il est parfois possible de le mettre sous une forme linéaire en l’exprimant différemment (ex.: X = Ln(x) ou X = exp(x)).
  + De plus, les méthodes de résolutions de systèmes linéaires par méthode matricielle (cela exclut celle itérative) sont habituellement beaucoup plus fiables que celle pour méthode non linéaire. On n’a notamment jamais parlé de la divergence des méthodes de résolution de système linéaire dans le cours, mais on en a souvent parlé pour celle non linéaire.

## Méthode de Gauss:

* + - Dans la Ti (bien qu’un peu différent comme résultat): ref (matrice).
    - La méthode de Gauss est une méthode de résolution de système linéaire qui transforme les équations de départ sous la forme d’une matrice triangulaire haut ou bas (ici = matrice triangulaire haute, comme celle de la Ti).
    - Pour ce faire, il suffit de faire l’opération suivante: [](http://api.gmath.guru/cgi-bin/gmath?%5Cdpi%7B480%7DL_n%5C%20%3D%5C%20L_n-%5Cleft(%5Cfrac%7BCoeff_%7Bpivot%5C%20de%5C%20L_n%7D%7D%7BCoeff_%7Bpivot%5C%20de%5C%20L_%7Bpivot%7D%7D%7D%5Cright)%5Ccdot%20L_%7Bpivot%7D)
    - Où les coefficients “pivot” sont de la même colonne et la ligne pivot est la ligne où on est rendu. Au départ c’est la ligne 1, puis la ligne 2 et ainsi de suite. À chaque fois la colonne du coefficient pivot a le même numéro que le numéro de la ligne pivot.
    - Le calcul de ce fait en O(n²) où O(n²) est une approximation asymptotique d’une série Arithmétique exécutée N fois (*je pense, date du début du cours d’Algorithme, ELE440*). Le calcul se fait donc assez lentement, mais plus vite qu’avec Gauss-Jordan (qui a le même ordre asymptotique, mais qui est vraiment un O(n²) qui n’a pas été arrondi de manière asymptotique).
    - **gauss\_mat(matrice)**:
      * Donne la démarche complète de la méthode de Gauss, inverse automatiquement et dit les lignes inversées lorsque pivot = 0 rencontré et retourne une matrice P qui, une fois multiplié avec le vecteur des variables de départ, permet d’avoir la position finale des variables après que les lignes ont été déplacées.
      * matrice : la matrice à résoudre sous le format utilisé à la main, soit dans le calcul matriciel A\*x = B où x = ?, matrice = [A,B]

## Méthode de Gauss-Jordan:

* + - Dans la Ti (donne le même résultat, mais sans la matrice P): rref (matrice).
    - La méthode de Gauss-Jordan effectue le même calcul que Gauss, mais à la différence qu’il commence par diviser les valeurs de sa ligne par la valeur de son pivot: [https://lh5.googleusercontent.com/FqJOA9qiHYub4ot4i7ElVemy8X8dBwdV9X56_WO8ruMMuARKAoSwov5f2vYkqz0OsQZ0yTjvgHEYD6FyAKEs_hiVR7EN7cxyOkiHF_U2G5Vrri4ziH2oPKH5HPnKRuVYwKi9r5Za](http://api.gmath.guru/cgi-bin/gmath?%5Cdpi%7B480%7DL_%7Bpivot%7D%5C%20%3D%5C%20%5Cleft(%5Cfrac%7B1%7D%7BCoeff_%7Bpivot%5C%20de%5C%20L_%7Bpivot%7D%7D%7D%5Cright)%5Ccdot%20L_%7Bpivot%7D)avant d’ensuite faire [](http://api.gmath.guru/cgi-bin/gmath?%5Cdpi%7B480%7DL_n%5C%20%3D%5C%20L_n-%5Cleft(%5Cfrac%7BCoeff_%7Bpivot%5C%20de%5C%20L_n%7D%7D%7BCoeff_%7Bpivot%5C%20de%5C%20L_%7Bpivot%7D%7D%7D%5Cright)%5Ccdot%20L_%7Bpivot%7D)sur toutes les lignes de la matrice (à chaque fois, au lieu de seulement celle plus basse), sauf elle-même.
    - La méthode a donc un ordre de temps asymptotique de O(n²), ce qui est assez lent, mais on a directement la réponse dans la dernière colonne au lieu de devoir, comme avec Gauss, calculer à la fin les coefficients en remontant la matrice (ce qui va se faire, en théorie, en O(n)).
    - **gauss\_jordan(matrice)**:
      * Donne la démarche complète de la méthode de Gauss-Jordan, inverse automatiquement et dit les lignes inversées lorsque pivot = 0 rencontré et retourne une matrice P qui, une fois multiplié avec le vecteur des variables de départ, permet d’avoir la position finale des variables après que les lignes ont été déplacées.
      * matrice : la matrice à résoudre sous le format utilisé à la main, soit dans le calcul matriciel A\*x = B où x = ?, matrice = [A,B]

## Méthode de résolution par la décomposition LU:

* + - Dans la Ti: la fonction “LU matrice, l, u,p” donne les matrices L, U et P de la décomposition LU de la matrice carrée A (A dans A\*x=B où x= ?), mais ne donne pas le résultat final X qui doit être manuellement fait et étrangement, certains tests ont montré que la décomposition de la Ti marchait mal, car ne revenait pas comme la matrice de départ (ce qui aurait dû être le cas). Encore là, c’est un test qui a donnée cela.
    - Note: On dit toujours décomposition LU, mais le terme exact serait “décomposition L, U, P”, car on a 3 matrices à la fin.
    - La méthode de résolution par la décomposition matricielle LU se fait en quelques étapes:
      * 1) À partir du système A\*x = B, on décompose la matrice A en 3 matrices: L (low), U (upper) et P. La matrice U est obtenue en faisant Gauss sur la matrice A et L contient la valeur de la division des coefficients qui ont été obtenus durant l’exécution de la méthode de Gauss. La matrice P est une matrice identité aux mêmes dimensions que A qui, lorsqu’on a inversé des lignes dû au pivot = 0 dans la matrice A, on aussi fait la même chose dans la matrice P. À la fin, on peut retrouver la matrice A en fessant L\*U\*P.
      * 2) A = L\*U et A\*x = B → L\*U\*x = B → y = U\*x, donc L\*y = B, on trouve y (j’ai utilisé Gauss-Jordan de la Ti).
      * 3) U\*x = y → On trouve x (j’ai utilisé Gauss-Jordan de la Ti).
      * 4) On multiplie x par P afin de repositionner les valeurs des X dans le même ordre qu’au début (l’ordre avant les permutations de ligne).
      * Note: Cette méthode suppose que la méthode de Gauss-Jordan de la Ti n’effectue pas d’échange de ligne durant son exécution. Pour un calcul implémenté dans un autre langage, utiliser sa propre méthode de Gauss-Jordan qui retourne aussi la matrice P de son calcul pour être 100 % sûr.
    - **calcul\_lu2(matrice)**:
      * Donne la démarche complète pour obtenir les matrices L et U et fournit aussi la matrice P.
      * Par la suite, calcul de la matrice Y qui permet d’obtenir la matrice X, soit ce qu’on recherche. La démarche complète pour y arriver est fournie par l’exécution du programme.
      * matrice : la matrice à résoudre sous le format pour les méthodes Gauss et Gauss-Jordan, soit dans le calcul matriciel A\*x = B où x = ?, matrice = [A,B].

## Valeurs propres (Eigenvalues):

* + - Fonction Ti: eigVl(matrice) → donne toutes les valeurs propres de la matrice (comme la fonction ci-dessous).
    - **eigenvalues(matrice)**:
      * Trouve les valeurs propres d’une matrice en faisant: solve(det(matrice - I\*λ)=0,λ).
      * La matrice doit être carrée.

* + **Vecteur propre (Eigenvector):**
    - Fonction Ti: eigVc(matrice) → donne tous les vecteurs propres de la matrice.
    - **eigenvectors(matrice, vecteur)**
      * Fonction qui va faire l’opération nécessaire pour vérifier si c’est un vecteur propre + va faire le calcul de eigenvalues pour vérification visuel par l’utilisateur dit quoi voir si la valeur affichée correspond à l’une des valeurs propres de la matrice.

## Décomposition en Valeurs Singulières - SVD (Singulars Values Decomposition)

* + - Est utilisés dans différents domaines tels que l’imagerie, la PCA, l’intelligence artificielle et bien d’autres domaines.
    - La décomposition en valeur singulière consiste à d’abord décomposer une matrice A (carré ou rectangulaire) comme étant le produit de 3 matrices: Q1, ∑ et Q2 (ce qui n’a pas été vu dans le cours). Par la suite, il est possible de reconstruire partiellement la matrice A tout en gardant uniquement les parties les plus importantes de cette matrice, ce qui est notamment très utile avec de très grosses matrices (comme une image de très haute qualité). On peut aussi décider de reconstruire entièrement la matrice A si on le souhaite.
    - **svd\_calculus (Q1, ∑, Q2, nombre\_iterations)**
      * Le programme construit étape par étape la matrice A via les 3 matrices et dit le résultat obtenu à chacune de ces étapes.
      * À la fin, le programme dit le % des informations du A de départ qui sont présent dans celle qui à été construite.
      * Q1 = la matrice orthogonale (donc carré) Q1 de dimension N\*N.
      * Q2 = la matrice orthogonale (donc carré) Q2 de dimension P\*P
      * ∑ = la matrice ∑ de dimension N\*P, qui sera carré si N = P.
      * nombre\_itération = le nombre de boucles à faire. Le nombre maximal qui sera effectué sera le nombre de lignes de la matrice ∑, soit N.

## Matrice inverse ou pseudo-inverse (si rectangulaire) à l’aide de la SVD

* + - À l’aide des matrices de la décomposition de en valeur singulière d’une matrice A, il est aussi possible d’obtenir sa matrice inverse, et ce, même si elle est rectangulaire. Par contre, si A est rectangulaire, on parle de pseudo-inverse.
    - **svd\_inverse (Q1, ∑, Q2, nombre\_iterations)**
      * Le programme dit à chaque étape de construction de la matrice inverse ce qu’il a fait et affiché à la fin le pourcentage des informations de départ qui ont été utilisé pour faire le calcul.
      * Q1 = la matrice orthogonale (donc carré) Q1 de dimension N\*N.
      * Q2 = la matrice orthogonale (donc carré) Q2 de dimension P\*P
      * ∑ = la matrice ∑ de dimension N\*P, qui sera carré si N = P.
      * nombre\_itération = le nombre de boucles à faire. Le nombre maximal qui sera effectué sera le nombre de lignes de la matrice ∑, soit N.

## Vérification pour savoir si une matrice est orthogonale

* + - Fait tout simplement: est-ce que matrice\*transpose(matrice) = matrice identité de dimension nb\_ligne \* nb\_ligne.
    - **orthogonal (matrice)**
      * retourne VRAI (true) si orthogonal, FAUX (false) sinon.

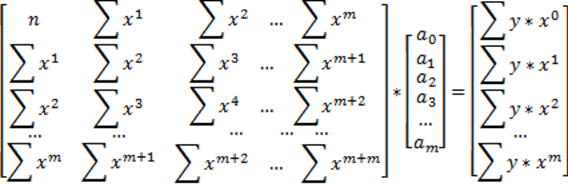
# Activité 3: Régression et Interpolation

* Définition Régression: La régression consiste à calculer un polynôme qui suit globalement (en moyenne) le plus que possible les points. Ceci est notamment utile si on veut voir si le point suit une forme de fonction particulière. L’équation de la fonction de régression est un polynôme d’ordre “m” de cette forme: https://lh3.googleusercontent.com/WuOnnKAecpziMQSdgjKX_GYsEiZeTehzkKbaTgZ-SpQ6Jmhpu3NyNG23vGBniMMuV-rosaueYgQVmb4FsFTxrJuvpFQ99uskY1aRmn7lViXF7uDL7KeBIWmgC9pQ4eUgjPbjlqreet l’erreur à minimiser correspond à : https://lh5.googleusercontent.com/ZkdOUNA56n_6irSgA-2DAq-30dIz2qJCo_oUjHN9vsbkFj9-y5tDgipL6rmJ9xY0Ucjqa0_lwq6ttxiuH0IEqZ0euDyRGYYtvQho8bw-dcX_Mg_wP9XoQvEN1eK7jsLFnQ99KLOToù “m” est le degré du polynombre et “n” est le nombre de points. À noter que si “m” = “n-1”, on aura la même équation avec l’interpolation. Cependant, l’objectif de la régression et de suivre la forme des points et non d’obtenir l’équation qui suit parfaitement les points. De plus, la performance de la résolution de la régression polynomiale à une performance comparable à la méthode “de base” de l’interpolation soit la méthode la moins performante.
  + Définition Interpolation: L’interpolation consiste à calculer un polynôme qui suit parfaitement l’ensemble des points connus dans le but d’approximer des valeurs entre ces points (ou autour de ces points). On peut aussi faire une interpolation afin de pouvoir passer d’un nuage de point à une fonction qui peut ensuite être résolue ou utilisée avec d’autres fonctions ou méthode de calcul (ex.: EDO, Intégrer, Dérivé, Équation Non-Linéaire, Système Linéaire, etc). À noter qu’il n’existe qu’un seul polynôme de degré (nombre de points -1) qui passe par une série de points spécifiques. C’est ce qu’on obtient avec les fonctions d’interpolation avec un polynôme (soient toutes les méthodes sauf celles des splines).
  + Note #1: Un tableau de style Excel a été mis dans le fichier pour pouvoir y mettre les valeurs en X et en Y des points. Les variables qui avait été définit était {xx, yy} et {no.xx, no.yy}. Toutes les méthodes d’interpolation et de régression peuvent être affichées dans une page graphique.
  + Note #2: Pour afficher les fonctions de régression et d’interpolation dans une page graphique, il suffit de remplacer “variable” par “x” pour la page graphique qui utilise toujours la forme “f(x) = …”. Lorsqu’on fait cela, la fonction va enregistrer dans une variable la valeur demandée et remplacer “variable” par un nom bizarre tapé de manière à avoir très peu de probabilité d’être entré par l’utilisateur. La démarche affichée aura donc un nom bizarre qui se répète souvent. À la fin de la résolution, la fonction ou le programme évalue sa solution à la valeur demandée, ce qui retourne le résultat de la valeur du polynôme évalué au point demandé.
  + Note #3: Comme il est présenté (rapidement) dans la fonction de régression linéaire, n’importe quel système ou équation Non-Linéaire peut souvent (pas toujours facilement, mais parfois oui) comme étant une équation linéaire. Ceci s’applique notamment ici. Une équation de forme y(x) = exp(x) peut être mis sous la forme linéaire via X = Ln(x), ce qui donne Y(x) = 1\*X + 0 où X = Ln(x) et Y = y. Dans le cas où l’on a en plus des variables (comme c’est le cas dans les exercices du manuel), cela permet de trouver la valeur des inconnues de la fonction linéaire via la régression linéaire. Le même principe peut être fait sur une équation de plusieurs variables non linéaire afin de le mettre sous la forme d’un polynôme (linéaire ou non) de plusieurs variable, ce qui permet de le résoudre par méthode matricielle de système linéaire (plus simple que système de Newton).

## Régression Linéaire d’équation Linéaire ou Non-Linéaire (*qui ont été mis sous la forme linéair*e)

* + - Comme il est présenté dans le cours et dans le manuel, on peut exprimer une fonction non linéaire sous la forme d’une équation linéaire de forme Y = a1\*X + a0 où Y = expression (y) et X = expression (x) où x et y sont les vraies valeurs de départ. Une fois que les coefficients a1 et a0 sont obtenus, on peut retrouver les valeurs des inconnues de la fonction non linéaire de départ (voir le manuel ou les documents du cours pour plus de détail)
    - **reg\_nl (x, y, variable)**
      * La fonction affiche les valeurs des sommations (S\_x, S\_y, S\_xx et S\_xy où S = sommation) qui sont nécessaire pour faire le calcul et va calculer les coefficients A1 et A0 dans F (variable) = A1\*variable + A0.
      * x et y = listes des coordonnées cartésiennes des points
      * variable = la variable à mettre dans la fonction qu’on retourne OU une valeur pour lequel on veut savoir le résultat de la régression linéaire à ce point (notamment si on veut l’afficher dans page graphique Ti).

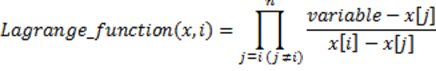
## Régression polynomiale

* + - La méthode que j’ai faite n’a pas été montrée en cours, mais a été faite pendant que le prof donnait le cours (Vincent Rougeau-Moss à un témoin). La méthode a été déduite de ce que le prof avait fait en généralisant le principe. Il a par la suite été vérifiée avec Wikipédia qui a confirmé que mon calcul était exactement cela, mais encore une fois pas montré explicitement comme étant cela.
    - La méthode utilisée est tout simplement la résolution à l’aide d’une matrice qui représente un système d’équations composé uniquement à l’aide des valeurs des points.
    - Cette matrice est déduite de la dérivée partielle de l’équation de l’erreur à minimiser en fonction des inconnues (*incertain, de mémoire*)
* Le système qui est résolu est le suivant:
  + - n = le nombre de points et les sommations sont par rapport à l’ensemble des n points.
    - m = le degré du polynôme demandé.
    - Afin d’éviter de recalculer plusieurs fois les valeurs des sommations, la fonction commence par mettre dans un tableau les valeurs des sommations (de n a sum(x^(m+m))) et de (sum(y) à sum(y\*x^m)). Par la suite, on met les valeurs dans la matrice qui sera ensuite résolue par Gauss-Jordan à l’aide de la fonction Ti “rref”. On ressort ensuite de cela les valeurs des inconnues qui seront ensuite utilisées pour construire la fonction du polynôme de la régression polynomiale.
    - **reg\_poly(x, y, variable, degré)**
      * x et y sont les listes des coordonnées cartésiennes des points.
      * variable = le nombre de la variable à mettre dans l’équation finale ou un chiffre si on veut directement évaluer la fonction obtenue à un point précis (notamment si on veut l’afficher)
      * degré = le degré du polynôme qu’on veut avoir, soit m dans la matrice.

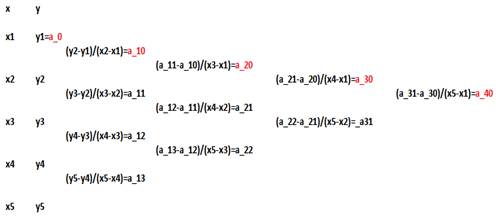
## Interpolation Polynomiale - Méthode la plus simple

* + - La méthode la plus simple pour faire une interpolation est de tout simplement trouver les valeurs des coefficients qu’il faut pour avoir pour obtenir le polynôme qui passe par les (n-1) points. Pour ce faire, on fait un système de (n) équations qu’on résout avec une des méthodes de résolution de système linéaire (j’ai pris Gauss-Jordan de la Ti).
    - C’est la pire méthode, mais c’est une méthode. Sa performance dépend principalement de celle qui résout la matrice et du temps pour construire la matrice.
    - **interpol\_poly(x, y, var)**
      * x et y sont les listes des coordonnées cartésiennes des points.
      * variable = le nombre de la variable à mettre dans l’équation finale ou un chiffre si on veut directement évaluer la fonction obtenue à un point précis (notamment si on veut l’afficher)

## Interpolation par polynôme de Lagrange

* + La méthode d’interpolation de Lagrange est une méthode assez simple qui se fait en deux boucles imbriquées. soit:  où 
    - Note: Bien que très simple, elle pose le problème (actuellement pas résolue) qu’il semble que la seule méthode de l’implémenter dans un langage non mathématique (sans variable symbolique) soit de faire ce calcul pour chacun des points à afficher ou à calculer, ce qui est vraiment mauvais en termes de performance (en O(n³), car n\*O(n²)).
    - **poly\_lagrange(x, y, variable)**
      * x et y sont les listes des coordonnées cartésiennes des points.
      * variable = le nombre de la variable à mettre dans l’équation finale ou un chiffre si on veut directement évaluer la fonction obtenue à un point précis (notamment si on veut l’afficher).

## Interpolation par polynôme de Newton:

* + - La méthode d’interpolation de Newton est une méthode qui permet d’obtenir le polynôme de Lagrange à l’aide d’une autre manière. En théorie, il est possible de faire en sorte que l’ajout de nouveau point ne nécessite pas de recalculer l’ensemble des coefficients, mais seulement ceux nécessaires pour l’obtention des coefficients des points ajoutés¸. ce qui réduit considérablement le temps de calcul en cas d’ajout de points.
    - La méthode qui à été utilisé dans la Ti est celle du manuel qui se fait en arbre binaire, ce qui permet une bien meilleure performance que la méthode de Lagrange, soit une performance en O(N\*log(N)) (avec log en base 2).
    - Cette méthode se fait très bien à la main et la fonction va afficher cette méthode avec beaucoup de texte. C’est la seule méthode où si n’as aucune idée de quoi ça parle, tu ne comprends pas ce que ça fait malgré la grande quantité de texte que ça affiche pour aider à comprendre. J’ai corrigé et peaufiné beaucoup l’affichage, mais ça reste qu’il faut savoir le calcul qu’elle fait pour bien comprendre.
    - Voici un exemple de l’arbre binaire qui est résolu avec 5 points
* 
  + - * À la fin de la résolution de cet arbre binaire, on a une matrice ayant l’ensemble des coefficients (soit ici {a0, a1, a2, a3, a4} ou {a0, a\_10, a\_20, a\_30, a\_40}) et il suffit de les multiplier par des groupes de coefficient de (x-xi), ce qui donne dans cet exemple: y(x) = a0 + a1\*(x-x1) + a2\*(x-x1)\*(x-x2) + a3\*(x-x1)\*(x-x2)\*(x-x3) + a4\*(x-x1)\*(x-x2)\*(x-x3)\*(x-x4). Une fois développer, on obtient la même équation qu’on aurait en développant le polynombre de Lagrange.
    - **poly\_newton (x,y, var)**
      * x et y sont les listes des coordonnées cartésiennes des points.
      * variable = le nombre de la variable à mettre dans l’équation finale ou un chiffre si on veut directement évaluer la fonction obtenue à un point précis (notamment si on veut l’afficher).

## Introduction aux splines

* + - Toutes les méthodes précédemment faites pour l’interpolation, soit la méthode de base, Lagrange et Newton, ont toutes un point faible en commun, soit le fait qu’il utilise un seul polynôme pour suivre l’ensemble des points.
    - Le problème avec cela est qu’avec les polynômes d’ordres élevés, on a souvent des “déformations” à proximité des points des extrémités, ce qui donne un aspect esthétique pas “super” et de très mauvaises approximations de valeur proche des points aux extrémités.
    - La solution: Utiliser plusieurs polynômes d’ordre faible sur les (n) points.
    - La méthode des splines consiste à utiliser (n-1) polynôme d’ordre faible pour les (n-1) intervalles de points.
    - Grâce à cela on obtient un graphique qui n’a jamais de déformations aux extrémités, mais pour les ordres de 2 et plus, la méthode vue en classe est vraiment pas performante dû à la résolution de matrice.
    - Les méthodes des splines vues en classe sont pour des polynômes d’ordre 1 (spline linéaire) , 2 (spline quadratique) et 3 (spline cubique). En examen, seul ordre 1 et 2 sont possible et on jamais eu à faire effectivement des splines d’ordre 3 à la main.
    - Donc graphiques plus beaux même avec beaucoup de points, meilleure approximation de valeur aux extrémités, mais plus longue à calculer.

## Spline Linéaire (ordre 1)

* + - Les splines linéaires sont les plus simples. Elles sont construites à l’aide du polynôme de Lagrange de degré 1, ce qui se fait en Θ(1), soit quasiment instantanément (pour le calcul d’un intervalle). On a donc globalement une performance en Θ(n-1) = Θ(n).
    - Pour ce faire, on utilise tout simplement le point de début et de fin de chaque intervalle de point, ce qui trace des droites entre les points.
    - Cette méthode est parfaitement suffisante si les points sont très rapprochés ou si on veut une interpolation qui s’effectue très rapidement.
    - **spline\_lin(x, y, var)**
      * x et y sont les listes des coordonnées cartésiennes des points.
      * variable = le nombre de la variable à mettre dans l’équation finale ou un chiffre si on veut directement évaluer la fonction obtenue à un point précis (notamment si on veut l’afficher).

## Spline Quadratique (ordre 2)

* + - Les splines quadratiques sont des polynômes d’ordre 2 (des paraboles) qui sont faits entre les (n) points, soit sur les (n-1) intervalles de point.
    - Puisqu’un polynombre d’ordre 2 est de la forme: y(x) = a\*x² + b\*x + c, soit 3 inconnues par équation, on doit résoudre un système d’équations linéaires composé de 3\*(n-1) = 3\*n-3 équations.
    - Le programme explique sa démarche de construction de cette matrice qui est la suivante:
      * La première condition que doivent avoir les polynômes est qu’ils doivent suivre les points de début et de fin des intervalles :
        + Pour le 1er et le dernier point: F [i] (xi) = y[i],
        + Pour les autres: F [i] (xi) = y[i] = F [i+1] (xi)
      * La deuxième condition est qu’aux intervalles de point, la dérivée de la fonction de l’intervalle d’avant et de l'intervalle d’après sont les mêmes afin d’avoir une continuité entre les intervalles:
        + Pour tous les points autre que le 1er et le dernier: F’[i](xi) = F’[i+1](xi) → F’ [i] (xi) - F’ [i+1] (xi) = 0.
      * La 3ème condition est une condition initiale, la valeur de l’un des coefficients, pour un des intervalles, doit être connue. Par défaut, on peut mettre que l’accélération initiale est 0, donc que F’’ [1] (x1) = 0.
    - À partir de cela on construit une matrice, on la résout par la méthode de Gauss-Jordan (ou une autre) et on obtient la valeur des coefficients.
    - Il suffit ensuite de redistribuer ces coefficients pour obtenir les fonctions entre chaque intervalle. Lorsqu’on veut évaluer la valeur à un point, il faut trouver à quel intervalle correspond ce point et ensuite évaluer la fonction de cet intervalle à ce point.
    - **spline\_quad(x, y, variable, string\_coeff\_CI, liste\_CI)** 
      * x et y sont les listes des coordonnées cartésiennes des points.
      * variable = le nombre de la variable à mettre dans l’équation finale ou un chiffre si on veut directement évaluer la fonction obtenue à un point précis (notamment si on veut l’afficher).
      * string\_coeff\_CI: Un string de la forme “[lettre\_coefficient][no\_intervalle\_de\_CI]”
        + [lettre\_coefficient]:

“a” si CI pour la dérivée second (F’’ ou l’accélération)

“b” si CI pour la dérivée première (F’ ou la vitesse)

“c” si CI pour la fonction elle-même (F ou la position)

* + - * + [no\_intervalle\_de\_CI]:

Le numéro de l'intervalle où la CI doit s’appliquer.

* + - * + Exemple:

“a1” = la condition initiale est pour F’’ [1] (x1)

“b3” = la condition initiale est pour F’ [3] (x3)

“c10” = la condition initiale est pour F[10] (x10)

* + - * liste\_CI: {x\_ci, y\_ci} où x\_ci = la valeur en X connue et y\_ci est la valeur de la fonction (F’’, F’ ou F) que cette fonction faut à ce point. Truc: mettre xx[no] où no = [no\_intervalle\_de\_CI].
      * Exemple: spline\_quad(xx, yy, “a1”, {xx[1], 0})
        + F’’[1] (xx[1]) = 0.

## Spline Cubique (ordre 3)

* + - Les splines cubiques sont construites de la même manière que les splines quadratiques, mais avec une condition supplémentaire due au fait qu’on a maintenant des polynômes d’ordre 3 entre les intervalles.
    - Les polynômes d’ordre 3 ont la forme: y(x) = a\*x³ + b\*x²+c\*x+d, on a donc 4 inconnues pour chacun des (n-1) intervalles, soit un total de 4\*(n-1) = 4\*n-4 équations linéaires à résoudre.
    - Pour ce faire, les polynômes doivent suivre 4 règles:
    - Condition #1: Les fonctions suivent l’ensemble des points connus;
      * 1er et dernier point: F[i] (x[i]) = y[i]
      * Pour les autres: F[i] (x[i]) = y[i] = F[i+1] (x[i])
    - Condition #2: Entre les intervalles, les dérivées premières des fonctions sont les mêmes afin d’être continue:
      * Pour tous les points sauf le 1er et le dernier: F’[i](xi) = F’[i+1](xi) → F’ [i] (xi) - F’ [i+1] (xi) = 0.
    - Condition #3: Entre les intervalles, les dérivées secondes des fonctions sont les mêmes afin d’être continue:
      * Pour tous les points sauf le 1er et le dernier: F’’[i](xi) = F’’[i+1](xi) → F’’ [i] (xi) - F’’ [i+1] (xi) = 0.
    - Condition #4: On connaît la valeur de deux coefficients parmi ceux des fonctions (peuvent être des fonctions différentes et des coefficients différents). Par défaut, on a les splines dites “naturelle”, qui ont l’accélération initiale et finale à 0.
    - **spline\_cub(x, y, liste\_string\_nom\_coeff\_CI, liste\_CI)**
      * Même fonctionnement que les splines quadratique, mais on a maintenant 2 string (au lieu d’une + mit dans une liste) et on a deux valeurs en x et en y (soit 4 valeurs) pour les CI (au lieu de 2 valeurs).
      * liste\_string = {string\_nom\_coeff\_CI1, string\_nom\_coeff\_CI2}
      * Liste\_CI = {x\_CI1, y\_CI1, x\_CI2, y\_CI2}
    - Note: la fonction n’est pas affichée (la page a été fermée), car #1) Par à l’examen, #2) Le simple fait de la garder ouverte prend environ 10 Ko, car environ 200 lignes de codes.
    - **Note importante**: **Une erreur a été identifié**e (après le cours) dans le code de la dernière étape (là où il y a une sommation) **mais n’a pas pu être corrigé** dû à l’impossibilité de résoudre le problème de la Ti qui est incapable de reconnaître que la condition donne un résultat TRUE/FALSE dans la condition de la fonction “get\_line\_CI”. L’erreur a par contre été corrigée dans le fichier Matlab de la fonction. Puisque cette fonction n’est pas demandée dans le cours (ne sera jamais à l’examen), ceci n’est pas si grave pour le cours. Dans le cas où on veut utiliser cette fonction, il faut utiliser la version Matlab ou la version de la Ti, mais avec des splines “naturelles” (par exemple, soit le cas où l’accélération initiale et finale est de 0 et donc, la ligne des 2 conditions initiales va rester à 0).

# Activité 4: Dérivé Numérique

* + Les méthodes de dérivée numériques se font toute en 1 seule ligne de calcul, d’où pourquoi elles font simplement dire d’aller voir les tableaux du manuel aux pages 317 à 319. La dernière méthode (soit la première qu’on voit dans le fichier Ti) exécute l’une de ces méthodes dérivées numériques (au choix) pour le degré de dérivation choisie, pour l’ensemble des points.
  + Les méthodes de dérivée d’ordre 1, 2, 3 et 4 n’ont pas de vérification de paramètre, mais celle pour l’ensemble des points s’assure d’avoir les bons paramètres et s’assurer de ne faire exécuter ces méthodes que sur les points qui peuvent être calculés à partir de cette méthode (dû au fait qu’on utilise les points suivants ou/et précédents d’un point pour calculer la dérivée à ce point).
  + Toutes les méthodes (sauf celle de tous les points) sont exactement pareilles en termes de paramètre. Il suffit de changer le no pour avoir la dérive suivante.
  + Un tableau a été mis à la disposition de l’utilisateur pour pouvoir y mettre les valeurs en x et en y. Les variables qui avait été utilisé sont {no.xx, no.yy}.

## Dérivé première numérique

* + - **diff\_1(x, y, pos\_xi, choix)**
      * x et y sont les coordonnées cartésiennes des points de la fonction.
      * pos\_xi = le numéro du point pour lequel on veut avoir la dérivée (un numéro entre 1 et le nombre de points).
      * choix = le numéro de la méthode de dérivation choisie, ce qui se réfère au manuel:
        + #1: 2 points vers l’avant
        + #2: 3 points vers l’avant
        + #3: 2 points en arrière
        + #4: 3 points en arrière
        + #5: 2 points centraux
        + #6: 4 points centraux

## Dérivé seconde numérique

* + - **diff\_2(x, y, pos\_xi, choix)**
      * x et y sont les coordonnées cartésiennes des points de la fonction.
      * pos\_xi = le numéro du point pour lequel on veut avoir la dérivée (un numéro entre 1 et le nombre de points).
      * choix = le numéro de la méthode de dérivation choisie, ce qui se réfère au manuel:
        + #1: 3 points vers l’avant
        + #2: 4 points vers l’avant
        + #3: 3 points en arrière
        + #4: 4 points en arrière
        + #5: 3 points centraux
        + #6: 5 points centraux

## Dérivé tertiaire numérique

* + - **diff\_3(x, y, pos\_xi, choix)**
      * x et y sont les coordonnées cartésiennes des points de la fonction.
      * pos\_xi = le numéro du point pour lequel on veut avoir la dérivée (un numéro entre 1 et le nombre de points).
      * choix = le numéro de la méthode de dérivation choisie, ce qui se réfère au manuel:
        + #1: 4 points vers l’avant
        + #2: 5 points vers l’avant
        + #3: 4 points en arrière
        + #4: 5 points en arrière
        + #5: 4 points centraux
        + #6: 6 points centraux

## Dérivé 4ème numérique

* + - **diff\_4(x, y, pos\_xi, choix)**
      * x et y sont les coordonnées cartésiennes des points de la fonction.
      * pos\_xi = le numéro du point pour lequel on veut avoir la dérivée (un numéro entre 1 et le nombre de points).
      * choix = le numéro de la méthode de dérivation choisie, ce qui se réfère au manuel:
        + #1: 5 points vers l’avant
        + #2: 6 points vers l’avant
        + #3: 5 points en arrière
        + #4: 6 points en arrière
        + #5: 5 points centraux
        + #6: 7 points centraux

## Plusieurs dérivées de suite

* + - **serie\_diff(x, y, choix, choix\_diff)**
      * La fonction vérifie si les paramètres sont valides et va construire une liste qui aura la valeur obtenue par la fonction “diff\_no” où ce no = choix\_diff.
      * Si c’est impossible pour un point (ex.: 2 points avant avec diff\_1 et le dernier point), alors elle met “undef” à cette place de la liste.
      * Les paramètres: x, y et choix correspondent à la fonction diff\_no choisit à l’aide de choix\_diff.

# Activité 5: Intégration numérique

* + Les méthodes d’intégration numériques présentées dans le cours ont toutes été regroupées dans une seule fonction due au fait qu’elle effectue tout un calcul qui s’écrit en une seule ligne.
  + De ce fait, comme avec les dérivés numériques, la fonction va afficher à l’utilisateur les numéros des pages du manuel à aller voir pour savoir ce qui a été fait.
  + Toutes les méthodes (sauf celles de Simpsons) sont faites pour avoir des espacements entre les points différents (au cas où le prof aurait décidé de nuire à tout le monde pour le plaisir en changeant juste un peu l’espacement entre deux points subtils). De ce fait, la performance est légèrement plus faible, dû au calcul de l’espacement à chaque fois, mais cela permet d’être utilisé sans problème sur n’importe quelle série de points.
  + Un tableur a été mis à la disposition de l’utilisateur pour pouvoir mettre ces valeurs des coordonnées des points. Les variables qui avait été utilisés sont {no.xx, no.yy}
  + **int\_num(x, y, fonction, variable, choix, is\_second)**
    - x et y = les listes des coordonnées cartésiennes des points, soit les les valeurs des points de l’intervalle où fait l’intégrale (et pas plus).
    - “fonction” et “variable” sont uniquement pour la méthode du point milieu, car il faut calculer la valeur du point entre deux intervalles de point connus. Mettre deux lettres à la place pour les autres méthodes.
    - “choix” et “is\_second” sont pour les différentes méthodes de calcul numérique:
      * Méthode des rectangles composites: choix = 1, is\_second = 0
      * Méthode du point Milieu (Midpt) composite: choix = 1, is\_second = 1
      * Méthode par Trapèze composite: choix = 2, is\_second = 0 ou 1
      * Méthode Simpson ⅓ (polynôme d’ordre 2): choix = 3, is\_second = 0.
        + Note: Le nombre d’intervalles (nb point-1) doit être un chiffre pair. et l’espacement entre les points doit être constant.
      * Méthode Simpson ⅜ (polynôme d’ordre 3): choix = 3, is\_second = 1.
        + Note: Le nombre d’intervalles (nb point - 1) doit être divisible par 3, donc (nb point-1) % 3 == 0.

# Activité 6: EDO et Système d’EDO d’ordre 1 à une ou plusieurs variables (ordre supérieur par encore fait)

* + Rappel définition EDO d’ordre 1 à 1 variable: Une EDO (ou ODE en anglais) d’ordre 1 à une variable est une équation de la forme [https://lh6.googleusercontent.com/otCAhdA2pYDQbkrzEv7RzJJd8W00_scZcmF5kZa-167Ov-JkaxK-ESwFoLdMo3TRgRpiXuHAa9yYlErL6Hx_tfqQRlBA-htEn5_-DaXxKlg7AUs6ZzCYVA8E4E0bgFezMF2W0bTO](http://api.gmath.guru/cgi-bin/gmath?%5Cdpi%7B480%7D%5Cfrac%7Bd%5Cleft(y%5Cright)%7D%7Bd%5Cleft(x%5Cright)%7D%5C%20%3D%5C%20f%5Cleft(x%2Cy%5Cright))où f(x,y) est la fonction de la dérivée qu’on connaît. Ce qu’on cherche, c’est savoir y(x) = ?.
  + Note importante: Il est important de garder en tête que dans les EDO, il y a toujours au moins une variable indépendante (x dans l’exemple précédent). Si tel n’est pas le cas, aucune des équations dans cette section ne sera applicable. Au besoin, pensé à l’affichage graphique ou le fait (exemple) d’avoir plusieurs test/no de fonction où ces numéros sont indépendants des fonctions, donc sont des variables indépendantes.
  + Rappel définition EDO d’ordre 1 à plusieurs variables: Une EDO d’ordre 1 à plusieurs variables est une équation de la forme [https://lh6.googleusercontent.com/Z3EC-l6PecVnrvuLE_fOpdsOMkU5Ra10Ve8oAi7nc_73eRH6XaUhaROfGlWPDXS2_Qi66lLTgZxXbFIx1dvk7EZMdwJ8Ut5RiCD20K0RhN2icIwLnVdCdV1QNMaBflQqkWadCn9w](http://api.gmath.guru/cgi-bin/gmath?%5Cdpi%7B480%7D%5Cfrac%7Bd%5Cleft(y%5Cright)%7D%7Bd%5Cleft(x%5Cright)%7D%5C%20%3D%5C%20f%5Cleft(x%2Cy%2Cz%2C%5C%20...%5Cright))où la fonction de la dérivée de dépend de plusieurs variables. On aura donc un système d’équations composé des autres fonctions de dérivée (z et les autres) qu’on connaît aussi. On va donc chercher à résoudre l’ensemble de ces fonctions (y, z et les autres).
  + Rappel définition EDO d’ordre N à 1 variable: Une EDO d’ordre N à une variable sera une équation de la forme [https://lh3.googleusercontent.com/cw5BAWnrYow_ofmxso8tsqi-HLU1k3j3gAoe4jny-vtGl1kqUm0Y1czqWx0brqpJD2Ef2nDj4PT5Q15hWm3jiqp61wUbf_HnCxlYUWDVroe6BgekvnNZrbB63D-DgE8oQ-bqWi0e](http://api.gmath.guru/cgi-bin/gmath?%5Cdpi%7B480%7D%5Cfrac%7Bd%5EN%5Cleft(y%5Cright)%7D%7Bd%5Cleft(x%5Cright)%5EN%7D%3Df%5Cleft(x%2Cy%2C%5Cfrac%7Bd%5E%7BN-1%7D%5Cleft(y%5Cright)%7D%7Bd%5Cleft(x%5Cright)%5E%7BN-1%7D%7D%2C%5C%20%5Cfrac%7Bd%5E%7BN-2%7D%5Cleft(y%5Cright)%7D%7Bd%5Cleft(x%5Cright)%5E%7BN-2%7D%7D%2C%5C%20...%5Cright))où la fonction de la dérivée Nème est connue et peut dépendre de x et de tous les ordres précédents de dérivée de la fonction. Ceci n’a pas encore été fait (et n’a pas été présenté dans le cours et est très peu détaillé dans le manuel). On imagine facilement qu’on peut extrapoler le même concept pour ordre N à plusieurs variable.
  + Rappel méthode Ti pour 1 EDO à 1 variable: exemple: desolve(y’ = x-(x\*y)/2 and y(1) = 1, x,y).

## EDO ordre 1 à 1 variable - Méthode d’Euler et du Point Milieu (Midpt)

* + - **ode\_euler (x, fonction, liste\_variables, liste\_CI, choix)**
      * x = liste des valeurs des coordonnées cartésiennes en X.
      * fonction = f(x,y) dans les images dans les rappelles, soit la fonction de la dérivée première qu’on connait.
      * liste\_variables = listes des deux variables, soit {variable indépendante, variable dépendante} ou {x,y}.
      * liste\_CI = liste des deux CI pour les deux variables, soit {x\_CI, y\_CI} ou {x0, y0}.
      * choix = le numéro de la méthode calcul numérique choisit:
        + Euler Explicite: 1
        + Euler Modifié (plus précis): 2
        + Méthode du Point Milieu (Midpt): 3

## EDO ordre 1 à 1 variable - Méthode de Runge-Kutta

* + - **runge\_kutta(x, fonction, liste\_variable, liste\_CI, liste\_coeff\_a, liste\_coeff\_b, liste\_coeff\_c, degree)**
      * x = liste des valeurs des coordonnées cartésiennes en X.
      * fonction = f(x,y) dans les images dans des sections “rappel”, soit la fonction de la dérivée première qu’on connaît.
      * liste\_variables = listes des deux variables, soit {variable indépendante, variable dépendante} ou {x,y}.
      * liste\_CI = liste des deux CI pour les deux variables, soit {x\_CI, y\_CI} ou {x0, y0}.
      * liste\_coefff\_a, liste\_coeff\_b et liste\_coeff\_c sont les valeurs des coefficients à utiliser dans le calcul, ce qui dépend de la méthode à utiliser. Une feuille d’éditeur mathématique contient déjà toutes les valeurs de coefficient a, b et c pour toutes les méthodes présentées dans le manuel soit:
        + Ordre 2:

Euler modifié

Point Milieu

Méthode de Heun

* + - * + Ordre 3:

Méthode Classique

Méthode de Nystrom

Méthode presque optimal (Nearly Optimal)

Méthode de Heun

* + - * + Ordre 4:

Méthode Classique

* + - * + Il suffit de taper (ex) “coeff\_a.” et vous voyez une liste des coefficients déjà enregistrer. Le même principe pour “coeff\_b.” et “coeff\_c.”
      * degré = le degré de la méthode de Runge-Kutta utilisé, soit le nombre de points entre le point actuel et le point suivant qui seront utilisés pour faire le calcul du point suivant (plus élevé = plus précis et plus long à calculer).

## Système d’EDO d’ordre 1 - Méthode d’Euler ou du Point Milieu (Midpt)

* + - **systeuler\_ordre1(x, liste\_fonctions, liste\_variables, liste\_CI, choix)**
      * Même principe qu’avec 1 fonction, mais avec plusieurs fonctions qui sont toutes interdépendantes. On doit donc calculer le point suivant de toutes les fonctions afin de pouvoir passer au point en X suivant.
      * Les paramètres sont les mêmes qu’avant, mais au lieu d’avoir une seule fonction, on a une liste de fonctions, plus de variables et plus de CI.
      * Le reste ne change pas.
      * On peut l’utiliser pour un système d’équations ou une seule équation, d’où la programmation en Matlab que j’ai effectuée qui ne contient que cette méthode (et non en plus celle pour seulement une fonction).
      * Note: N’était pas à l’examen final.

## Système D’EDO d’ordre 1 - Méthode de Runge-Kutta

* + - **syst\_rk\_ordre1(x, liste\_fonctions, liste\_variables, liste\_CI, liste\_coeff\_a, liste\_coeff\_b, liste\_coeff\_c, degré)**
      * Les paramètres sont les mêmes par rapport à la fonction de Runge-Kutta pour une fonction à 1 variable, à la différence qu’on a une liste de fonction au lieu d’une seule fonction, plus de variables et plus de CI. Les coefficients a, b et c ne change pas et degré est comme avant.
      * On peut l’utiliser pour un système d’équations ou une seule équation, d’où la programmation en Matlab que j’ai effectué qui ne contient que cette méthode.
      * Note: N’était pas à l’examen final.

La seule fonction qui manque, même si pas encore faite et pas vue dans le cours, mais présentée dans le manuel, serait la résolution de :

1. Une EDO à une variable d’ordre N
2. Un système d'EDO à plusieurs variable d’ordre N (pour toutes les fonctions)
3. Un système d'EDO à plusieurs variables où chaque fonction peut avoir un ordre différent.

Note: À déjà fait système d’EDO d’ordre 2 avec 3 EDO interdépendante dans Matlab pour un projet personnelle. Pourrais éventuellement utiliser le même principe pour les 3 idées plus haut. (Vincent Rougeau-Moss)