Московский авиационный институт

(Национальный исследовательский университет)

Институт №8 «Компьютерные науки и прикладная математика»

Кафедра вычислительной математики и программирования

Курсовая работа

по курсу «Численные методы»

Вариант 2

Выполнил: Баранов И.В.

Группа: М8О-402Б-20

Преподаватель: доц. Пивоваров Д.Е.

Дата:

Оценка:

Москва, 2023

Оглавление

Введение

Метод бисопряженных градиентов — это итерационный численный метод решения систем линейных алгебраических уравнений (СЛАУ) крыловского типа. Он является обобщением метода сопряженных градиентов, который применяется для решения СЛАУ с самосопряженными (симметричными) матрицами. Метод бисопряженных градиентов позволяет решать СЛАУ с несимметричными матрицами, но он неустойчив и может сходиться медленно или вообще расходиться. Поэтому были разработаны различные модификации и улучшения этого метода, такие как стабилизированный метод бисопряженных градиентов, метод бисопряженных градиентов с чебышевскими полиномами, метод бисопряженных градиентов с перезапусками и другие. Эти методы имеют разную скорость сходимости, устойчивость и требования к памяти и вычислительным ресурсам.

Метод бисопряженных градиентов и его модификации широко используются в различных областях науки и техники, где возникают большие разреженные СЛАУ, например, в задачах оптимизации, численного решения дифференциальных уравнений, обработки сигналов, вычислительной физики и химии и других.

В данной курсовой работе мы будем рассматривать решение систем линейных алгебраических уравнений с несимметричными разреженными матрицами большой размерности при помощи стабилизированного метода бисопряженных градиентов.

Основная часть

Теоретические сведения

Стабилизированный метод бисопряжённых градиентов (англ. Biconjugate gradient stabilized method, BiCGStab) — итерационный метод решения СЛАУ крыловского типа. Разработан Ван дэр Ворстом для решения систем с несимметричными матрицами. Сходится быстрее, чем обычный метод бисопряженных градиентов, который является неустойчивым, и поэтому применяется чаще.

Описание алгоритма

Для решения СЛАУ вида Ax = b, где A – комплексная матрица, стабилизированным методом бисопряжённых градиентов может использоваться следующий алгоритм.

Подготовка перед итерационным процессом

Шаг 1. Выберем начальное приближение x^0.

Шаг 2. $r^0 = b - Ax^0$.

Шаг 3. $\tilde{r} = r^0$.

Шаг 4. $\rho^0 = \alpha^0 = \omega^0 = 1$.

Шаг 5. $v^0 = p^0 = 0$.

k-я итерация метода

Шаг 1. $\rho^k = (\tilde{r}, r^{k-1})$.

Шаг 2. $\beta^k = \frac{p^k}{p^{k-1}} \cdot \frac{\alpha^{k-1}}{\omega^{k-1}}$.

Шаг 3. $p^k = r^{k-1} + \beta^k (p^{k-1} - \omega^{k-1} v^{k-1})$.

Шаг 4. $v^k = Ap^k$.

Шаг 5. $\alpha^k = \frac{\rho^k}{(\tilde{r}, v^k)}$.

Шаг 6. $s^k = r^{k-1} - \alpha^k v^k$.

Шаг 7. $t^k = As^k$.

Шаг 8. $\omega^k = \frac{[t^k, s^k]}{[t^k, t^k]}$.

Шаг 9. $x^k = x^{k-1} + \omega^k s^k + \alpha^k p^k$.

Шаг 10. $r^k = s^k - \omega^k t^k$.

Критерий остановки итерационного процесса

Кроме традиционных критериев остановки, как число итераций $(k \leq k_{\max})$ и заданная невязка $\left(\frac{\|r^k\|}{\|b\|}\right) < \varepsilon$, так же остановку метода можно производить, когда величина $|\omega^k|$ стала меньше некоторого заранее заданного числа ε_ω.

Программная реализация

```
# Определим функцию для получения матрицы из файла
def get_matrix(filename, is_diag):
    # Откроем файл для чтения
    with open(filename) as f:
        # Прочитаем первую строку файла как размер матрицы
        shape = int(f.readline())
        # Создадим матрицу из оставшихся строк файла, преобразуя числа во
float
        matrix = [[float(num) for num in line.split()]
                  for _, line in zip(range(shape), f)]
        # Проверим, является ли матрица диагональной
```

```
        if is_diag:
            # Добавим нули на первую и последнюю позиции главной диагонали
            matrix[0].insert(0, 0)
            matrix[-1].append(0)
            # Разделим матрицу на три диагонали a, b, c
            a, b, c = zip(*matrix)
            # Создадим разреженную матрицу из диагоналей с помощью функции
diags
            matrix = diags([a[1:], b, c[:-1]], [-1, 0, 1])
            # Преобразуем матрицу в формат csc для эффективных вычислений
            matrix = csc_matrix(matrix)
        else:
            # Если матрица не диагональная, то просто Преобразуем ее в формат
csc
            # matrix = np.array([np.array(xi) for xi in matrix])
            matrix = csc_matrix(matrix)
        # Прочитаем последнюю строку файла как вектор b
        b = np.array([float(num) for num in f.readline().split()])
        # Вернем матрицу и вектор b
        return matrix, b


# Определим класс для решения системы линейных уравнений методом BiCGStab
class BiCGStab:
    # Определим конструктор класса с параметрами
    def __init__(self, matrix, b, output_file, log_file,
                 x0=None, eps=1e-5):
        # Установим имя файла для вывода результата
        self.output = 'res_default' if output_file is None else output_file
        # Установим имя файла для записи лога
        self.log = 'log_default' if log_file is None else log_file
        # Установим матрицу A
        self.matrix = matrix
        # Установим вектор b
```

```
        self.b = b
        # Установим точность решения
        self.eps = eps
        # Установим размер матрицы
        self.shape = matrix.shape[0]
        # Установим начальное приближение x0
        self.x0 = np.array([0] * self.shape) if x0 is None else x0
        # Установим счетчик итераций
        self.k = 0

    # Определим метод для решения системы
    def solve(self):
        # Откроем файл для записи лога
        f = open(self.log, 'w')
        # Вычислим начальный невязку r0
        r0 = self.b - self.matrix @ self.x0
        # Сохраним начальное приближение x0
        x0 = self.x0
        # Сохраним начальный невязку r2
        r2 = r0
        # Инициализируем параметры rho0, alpha0, omega0
        rho0 = 1
        alpha0 = 1
        omega0 = 1
        # Инициализируем векторы v0 и p0
        v0 = np.array([0] * self.shape)
        p0 = np.array([0] * self.shape)
        # Запишем начальные значения в лог
        f.write(f'r^0 = {self.b} - A @ {self.x0} = {r0}\n')
        f.write(f'\\tilde{{r^0}} = {r0} \n')
        f.write(f'\\rho^0 = \\alpha^0 = \\omega^0 = 1 \n')
        f.write(f'\\v^0 = \\p^0 = 0 \n')
        # Начать цикл решения
```

```
        while True:
            # Вычислим параметр rho
            rho = r2 @ r0
            # Вычислим параметр beta
            beta = (rho * alpha0) / (rho0 * omega0)
            # Вычислим вектор p
            p = r0 + beta * (p0 - omega0 * v0)
            # Вычислим вектор v
            v = self.matrix @ p
            # Вычислим параметр alpha
            alpha = rho / (r2 @ v)
            # Вычислим вектор s
            s = r0 - alpha * v
            # Вычислим вектор t
            t = self.matrix @ s
            # Вычислим параметр omega
            omega = (t @ s) / (t @ t)
            # Вычислим приближение x
            x = x0 + omega * s + alpha * p
            # Вычислим невязку r
            r = s - omega * t

            # Запишем текущие значения в лог
            f.write(f'Iter: {self.k}\n')
            f.write(f'\\rho^k = ({r2}, {r0}) = {rho}\n')
            f.write(f'\\beta^k = ({rho} * {alpha0}) / ({rho0} * {omega0}) =
{beta}\n')
            f.write(f'p^k = {r0} + {beta} * ({p0} - {omega0} * {v0}) =
{p}\n')
            f.write(f'v^k = {v}\n')
            f.write(f'\\alpha^k = {rho} / ({r2}, {v}) = {alpha}\n')
            f.write(f's^k = {r0} - {alpha} * {v} = {s}\n')
            f.write(f't^k = {t}\n')
            f.write(f'\\omega^k = ({t} , {s}) / ({t} , {t}) = {omega}\n')
```

```
        f.write(f'x^k = {x0} + {omega} * {s} + {alpha} * {p} = {x}\n')
        f.write(f'r^k = {s} - {omega} * {t} = {r}\n')
        f.write('=' * 10 + '\n')

        # Увеличим счетчик итераций
        self.k += 1
        # Проверим условие остановки
        if norm(r) < self.eps:
            break
        # Обновим значения для следующей итерации
        r0 = r
        rho0 = rho
        alpha0 = alpha
        omega0 = omega
        v0 = v
        p0 = p
        x0 = x
    # Закроем файл лога
    f.close()
    # Вернем решение x
    return x
  # Определим функцию для решения системы с предобуславливателем
  def precond_solve(self):
        # Открыть файл для дополнения лога
        f = open(self.log, 'a')
        # Вычислим разложение LU матрицы A с помощью функции spilu
        k_mtrx = spilu(self.matrix)
  # Вычислим начальную невязку r0
  r0 = self.b - self.matrix @ self.x0
# Сохраним начальное приближение x0
x0 = self.x0
# Сохраним начальную невязку r2
r2 = r0  # r_tilda
```

```
# Сохраним начальный вектор p
p = r0
# Начать цикл решения
while True:
    # Проверим, не равно ли скалярное произведение r0 и r2 нулю
    if r0 @ r2 == 0:
        # Запишем ошибку в лог и выйти из программы
        f.write(f'Error\nIters = self.k')
        exit()
    # Решить систему Kp* = p с помощью метода solve предобуславливателя
    pp = k_mtrx.solve(p)
    # Запишем результат в лог
    f.write('Kp* = p solved\n')
    # Вычислим произведение матрицы A и вектора pp
    tmp = self.matrix @ pp
    # Вычислим параметр alpha
    alpha = (r0 @ r2) / (tmp @ r2)
    # Вычислим вектор s
    s = r0 - alpha * tmp
    # Проверим, не меньше ли норма вектора s заданной точности
    if norm(s) < self.eps:
        # Увеличить счетчик итераций
        self.k += 1
        # Вычислим приближение x
        x = x0 + alpha * pp
        # Прервать цикл
        break
    # Решить систему Kz = s с помощью метода solve предобуславливателя
    z = k_mtrx.solve(s)
    # Запишем результат в лог
    f.write('Kz = s solved\n')
    # Вычислим произведение матрицы A и вектора z
    tmp2 = self.matrix @ z
```

```
        # Вычислим параметр omega
        omega = (tmp2 @ s) / (tmp2 @ tmp2)
        # Вычислим приближение x
        x = x0 + alpha * pp + omega * z
        # Вычислим невязку r
        r = s - omega * tmp2
        # Вычислим параметр beta
        beta = (r @ r2) / (r0 @ r2) * (alpha / omega)
        # Вычислим вектор p
        p = r + beta * (p - omega * tmp)
        # Увеличить счетчик итераций
        self.k += 1
        # Проверим, не меньше ли норма вектора r заданной точности
        if norm(r) < self.eps:
            # Прервать цикл
            break
        # Обновить значения для следующей итерации
        x0 = x
        r0 = r

        # Запишем текущие значения в лог
        f.write(f'Iter: {self.k}\nx0 = {x0}\np* = {pp}\nbeta = {beta}\n')
        f.write(f'z = {z}\nomega = {omega}\nr2 = {r2}\nr0 = {r0}')
        f.write('\n' + '=' * 10 + '\n')

    # Закрыть файл лога
    f.close()
    # Вернуть решение x
    return x

# Определим функцию для вывода решения в файл
def print_solution(self):
    # Засечь время начала решения
```

```
start = time()
# Вызвать метод solve для решения системы
x = self.solve()
# x = self.precond_solve()
# Засечь время окончания решения
end = time()
# Засечь время начала решения с помощью NumPy
start2 = time()
# x2 = bicgstab(self.matrix, self.b, tol=self.eps, x0=self.x0)[0]
# Решить систему с помощью функции np.linalg.solve
x2 = np.linalg.solve(self.matrix.toarray(), self.b)
# Засечь время окончания решения с помощью NumPy
end2 = time()
# Открыть файл для записи результата
with open(self.output, 'w') as f:
    # Запишем решение, полученное методом BiCGStab
    f.write('Наше решение:\n')
    f.write(f'{x.round(5)}\n')
    # Запишем точность решения
    f.write(f'EPS = {self.eps}\n')
    # Запишем размер матрицы
    f.write(f'Размерность матрицы = {self.shape}\n')
    # Запишем количество итераций
    f.write(f'Число итераций = {self.k}\n')
    # Запишем среднее значение решения
    f.write(f'Значение = {np.mean(x)}\n')
    # Запишем время решения
    f.write(f'Время = {round(end - start, 5)} sec\n')
    # Запишем решение, полученное с помощью NumPy
    f.write('\nРешение NumPy:\n')
    f.write(f'{x2.round(5)}\n')
    # Запишем среднее значение решения
    f.write(f'Значение = {np.mean(x2)}\n')
```

```
        # Запишем время решения
        f.write(f'Время = {round(end2 - start2, 5)} sec\n')
def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('--input', required=True, help='Input file')
    parser.add_argument('--output', required=True, help='Output file')
    parser.add_argument('--log', help='Log file')
    parser.add_argument('--eps', type=float, help='Epsilon', default=1e-2)
    parser.add_argument('--diag', help='If matrix is diag',
action='store_true')
    args = parser.parse_args()

    matrix, b = get_matrix(args.input, args.diag)

    solver = BiCGStab(matrix, b, output_file=args.output,
                      log_file=args.log, eps=args.eps)
    solver.print_solution()
if __name__ == "__main__":
    main()
```

Пример решения

- **для матрицы размерности 6**

```
Наше решение:
[-25.30795   1.26913   3.5461   10.6383   16.79731   6.53229]
EPS = 0.01
Размерность матрицы = 6
Число итераций = 6
Значение = 2.2458628793076536
Время = 0.03122 sec

Решение NumPy:
[-25.30795   1.26913   3.5461   10.6383   16.79731   6.53229]
Значение = 2.2458628841607555
```

```
Время = 0.0 sec
```

- **для матрицы размерности 45**

```
Наше решение:
[ 1.867414e+01  6.322890e+00  2.514286e+01  9.174003e+01  6.179489e+01
  1.897969e+01  3.089167e+01  4.931762e+01  4.654216e+01 -1.815079e+01
 -1.160692e+01 -1.283750e+01 -3.941360e+00 -3.206319e+01 -6.565930e+00
  9.333860e+00  1.772880e+00  8.378500e-01  6.716550e+00  5.530020e+00
 -2.611690e+00  3.022300e-01 -1.559000e-02 -1.171160e+00 -2.121730e+00
  8.119700e-01 -3.546200e-01  9.251000e-02  1.048900e-01  6.164800e-01
 -2.826900e-01  2.122900e-01 -1.044300e-01  4.310000e-02 -1.362400e-01
  1.071900e-01 -1.084100e-01  7.033000e-02 -3.640000e-02  1.496000e-02
 -3.794000e-02  4.467000e-02 -3.276000e-02  1.637000e-02  4.300000e-03]
EPS = 0.01
Размерность матрицы = 45
Число итераций = 16
Значение = 6.307979507345554
Время = 0.2343 sec

Решение NumPy:
[ 1.867453e+01  6.322750e+00  2.514339e+01  9.173987e+01  6.179495e+01
  1.897912e+01  3.089106e+01  4.931809e+01  4.654140e+01 -1.814997e+01
 -1.160736e+01 -1.283783e+01 -3.939670e+00 -3.206340e+01 -6.567380e+00
  9.335190e+00  1.773080e+00  8.360400e-01  6.716860e+00  5.530880e+00
 -2.612640e+00  3.025000e-01 -1.508000e-02 -1.170960e+00 -2.122430e+00
  8.128700e-01 -3.553500e-01  9.275000e-02  1.045000e-01  6.169300e-01
 -2.834800e-01  2.133000e-01 -1.050700e-01  4.330000e-02 -1.362900e-01
  1.077500e-01 -1.092800e-01  7.094000e-02 -3.632000e-02  1.481000e-02
 -3.822000e-02  4.515000e-02 -3.308000e-02  1.625000e-02  4.420000e-03]
Значение = 6.307973713940822
Время = 0.0 sec
```

- **для матрицы размерности 10**

```
Наше решение:
```

```
[-26.42468  -7.25845  11.40364  25.90414  32.72361   8.23053 -18.31816
 -21.06376  -1.5722    0.     ]
EPS = 0.01
Размерность матрицы = 10
Число итераций = 13
Значение = 0.3624649620456557
Время = 0.0781 sec

Решение NumPy:
[-26.42468  -7.25845  11.40364  25.90414  32.72361   8.23053 -18.31816
 -21.06376  -1.5722   -0.     ]
Значение = 0.3624652452221465
Время = 0.0 sec
```

Вывод

В ходе выполнения курсовой работы, мной были изучены методы решения СЛАУ с несимметричными разреженными матрицами большой размерности. Метод, рассматриваемый в работе, сходится быстрее, чем обычный метод бисопряженных градиентов. Таким образом, стабилизированный метод биспоряженных градиентов позволяет решать сложные задачи большой размерности за наименьшие время и количество шагов.

Список литературы

[1] Wiki. Стабилизированный метод бисопряжённых градиентов

https://ru.wikipedia.org/wiki/Стабилизированный_метод_бисопряжённых_градиентов

[2] Van der Vorst, H. A. (1992). "Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems".

[3] AlgoWiki. Стабилизированный метод бисопряженных градиентов (BiCGStab)

https://algowiki-project.org/ru/%D0%A1%D1%82%D0%B0%D0%B1%D0%B8%D0%BB%D0%B8%D0%B7%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D1%8B%D0%B9_%D0%BC%D0%B5%D1%82%D0%BE%D0%B4_%D0%B1%D0%B8%D1%81%D0%BE%D0%BF%D1%80%D1%8F%D0%B6%D0%B5%D0%BD%D0%BD%D1%8B%D1%85_%D0%B3%D1%80%D0%B0%D0%B4%D0%B8%D0%B5%D0%BD%D1%82%D0%BE%D0%B2_(BiCGStab)